Project ifchk: Host Based Promiscuous Mode Detection and Handling

Joshua Birnbaum
Noorg, Inc.
engineer@noorg.org
www.noorg.org/ifchk

Richard L. Kline
Computer Science Department
Pace University
New York, NY USA
rkline@pace.edu

ABSTRACT

In the face of mounting threats to operating system and network security, proactivity is fast becoming a necessity as opposed to a retrofitted afterthought. If made a central component of site security policy, proactivity can go a long way towards thwarting unauthorized access and usage of computing resources. The timely application of operating system patches, knowledge of current security related vulnerabilities, frequent review of system log output along with the use of well chosen security tools are just some examples of how modern computing environments can withstand determined attacks. **ifchk** (interface check) is a security tool for network interface promiscuous mode detection, interface management and traffic trend analysis. Written in the C programming language and initially released under IRIX, an SVR4 based Unix implementation from Silicon Graphics, Inc. (SGI), **ifchk** is now in the final stages of a porting effort to Linux.

1. INTRODUCTION

With the number of organizations and individuals participating in the networked world growing, there is a heightened emphasis on maintaining the operational integrity of such a communications infrastructure. Within the context of system and network administration, such efforts are often a mix of identifying security threats and maintaining acceptable levels of operational performance. Sometimes, one is linked to the other. In other cases, it is not. **ifchk** is a tool that addresses both needs.

In this paper, we begin by elaborating on the motivations behind the creation of **ifchk** followed by a discussion concerning network interface promiscuous mode operation. Program functionality and an examination of program implementation issues then follow. This leads us into how the **ifchk** core is structured from an algorithmic point of view. We then review related work and conclude by discussing future directions of **ifchk** development.

2. MOTIVATION AND BACKGROUND

There were three main motivators behind the writing of the **ifchk** security tool. Firstly, the program author is a long time user of open source software and wanted to respond in kind by giving something back to the open source community. Secondly, the author is an experienced Unix system administrator who saw his entry into systems programming as a powerful method to further explore Unix and Linux operating system implementation. To this end, **ifchk** has paid off handsomely. This exploration was further enhanced by the porting effort, mentioned above, of **ifchk** from IRIX on SGI to Linux. Differences in OS implementation, as they relate to **ifchk**, allowed for the examination of different application programming interfaces (API's) on both systems. Thirdly, it is hoped that **ifchk** can serve as a learning aid for individuals who wish to engage themselves in systems programming. Given that the **ifchk** source code is freely available, it is hoped that such individuals will take advantage of the ability to review code and learn from it.

2.1 Promiscuous Mode Operation

Here, we begin our discussion of the theoretical underpinnings of the **ifchk** tool which will serve to create a contextual foundation for what is to come. What is discussed here is within the context of TCP/IP protocol stack (Figure 1) operations and how they relate to core **ifchk** functions.

Data link implementations (*e.g.*, ATM, Ethernet, etc), as part of their design, will typically encapsulate the data that they transmit within implementation specific units. These units of data bear implementation specific names. For ex-

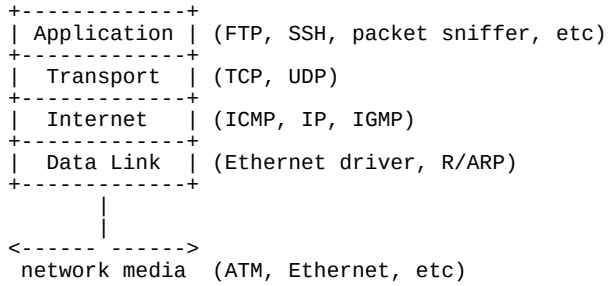

Figure 1: The four layers of the TCP/IP protocol stack along with what can be found at each layer.

ample, Asynchronous Transfer Mode (ATM) refers to units of data as **cells** while Ethernet refers to them as **frames**. (These units of data are often generically referred to as packets). This paper will concern itself primarily with Ethernet and Ethernet frames as the majority of networks are Ethernet based networks.

Upon receiving a frame, a network interface must make some decisions as to how it is to process that frame. If the frame was not corrupted in transit, (a determination made by the network hardware via a CRC or Cyclic Redundancy Check) it is handed off to the interface device driver at the data link layer of the TCP/IP protocol stack. What happens next is a function of frame destination address (ARP/MAC address) and whether the interface is in promiscuous mode or not. An interface running in non-promiscuous mode will only read frames specifically addressed to its ARP/MAC address (*e.g.*, 08:00:20:4b:1e:24), the network broadcast address (*e.g.*, ff:ff:ff:ff:ff:ff) and, if the interface is part of a multicast group, the multicast group address (*e.g.*, 01:00:5e:00:40:20). All other frames are ignored and, as a result, will not be copied from the network and passed up the protocol stack for further processing.

In contrast to the above, an interface running in promiscuous mode is interested in ALL frames traversing the network medium, and will pass every frame it receives up the protocol stack regardless of its destination ARP/MAC address (often for further processing by a packet sniffer such as as ethereal [3], tcpdump [10], etc.).

2.2 Examples of Packet Sniffer Usage

Packet sniffing is a facility that allows us to read data directly from the network as it is in transit between systems. The following examples illustrate packet sniffer usage:

- To pinpoint and debug network congestion/latency related problems stemming, for example, from faulty network hardware (*e.g.*, malfunctioning Ethernet switches), misbehaving network services (*e.g.*, slow NFS file service performance) or denial of service attacks such as ICMP broadcast storms.
- Reverse engineering. It is possible to gain a behavioral understanding of how a networked application functions, at the network protocol level, by observing its interaction with the network. For example, the Samba project made use of packet sniffing to observe how systems running Microsoft Windows communicated with one another with a view towards the creation of a viable file sharing utility compatible with both Windows and Unix/Linux systems.
- Password and data theft (unfortunately). Not all networks are switched and RSH (Remote SHell) and Telnet, as opposed to SSH, still find usage on both internal and external networks. A packet sniffer can read unencrypted RSH and Telnet application data with the result that user keystrokes between the RSH or Telnet client and server can be read verbatim.

Packet sniffing is somewhat stigmatized in that it is often associated with less than honorable uses, as described above. While packet sniffing does facilitate this kind of activity, the ability to read raw network frames directly from the network medium as an aid in the debugging of legitimate network related problems is a very powerful facility, as we have shown in the examples above.

3. IFCHK FUNCTIONALITY

Before discussing **ifchk** functions, an examination of network interface configuration under Unix and Linux is in order. ifconfig, a standard administrative tool, is provided by all vendors as part of the base operating system installation procedure. The assignment of unicast, broadcast and multicast IP addresses to interfaces, the modification of routing metrics, the creation of interface aliases and the enabling or disabling of interfaces are examples of common ifconfig tasks. ifconfig will also display status and state information for each interface (Figure 2).

The first line of output begins with an interface name. This is a name that references a given network interface on the system. On this Silicon Graphics workstation, `ec0` is the primary (referenced by the 0 in `ec0`) Ethernet interface. The `flags=d63 <UP, BROADCAST,...>` string following `ec0` are the status and state flags for the interface. It is these two pieces of data, the interface name and the flags output that we are most interested in. Here, `ec0` is in promiscuous mode as indicated by the displaying of the `PROMISC` flag. The second line of ifconfig output describes the IP address, subnet mask and broadcast address for interface `ec0`.

With the above foundation in place, we will now provide a description of **ifchk** functionality. The program performs the following functions:

```
ec0: flags=d63<UP,BROADCAST,NOTRAILERS,RUNNING,PROMISC,FILTMULTI,MULTICAST>
        inet 192.0.2.2 netmask 0xffffff00 broadcast 192.0.2.255
```

Figure 2: ifconfig output showing interface status and state.

1. **ifchk** will report on the state (`normal`, `*down*`, `PROMISC`, `PROMISC [*]`) of each interface attached to the system.

   (a) The state `normal` refers to an interface that is up. It is reading from and writing data to the network and is not in promiscuous mode.

   (b) The state `*down*` refers to an interface that is down. The system will not attempt to transmit data over an interface in this state.

   (c) The state `PROMISC` refers to an interface that is up. It is reading from and writing data to the network and IS in promiscuous mode.

   (d) The state `PROMISC [*]` refers to an interface that has been shutdown because **ifchk** was told, by the user invoking the program, to shutdown any interfaces found in promiscuous mode. The interface then enters into the `*down*` state described above.

2. **ifchk** will shutdown all interfaces running in promiscuous mode, if told to do so.

3. **ifchk** will report per-interface traffic metrics to help identify spikes in network traffic flow that may warrant further investigation. This is similar to output generated by the netstat command. netstat, like ifconfig, is standard on unix and linux systems and displays network status information such as the contents of the in-kernel routing table, integer counters describing both ingress and egress packet counts and per-protocol (TCP, UDP, ICMP, etc) statistics.

4. **ifchk** logs everything that it finds via syslogd (the Unix/ Linux system event logging daemon).

Console output from **ifchk** Linux (Figure 3) begins with a count of interfaces present on the system and then proceeds to describe the status of each.

```
interface(s): 6
     lo: normal
   eth0: PROMISC [*]
  bond0: *down*
   gre0: *down*
  tunl0: normal
 dummy0: *down*
```

Figure 3: ifchk output under Linux.

Console output from **ifchk** IRIX on a Silicon Graphics system with two interfaces (Figure 4) shows a dump of per-interface ingress/ egress packet count information. `Name` refers to an interface name (*e.g.*, `ec0`). `Index` refers to a kernel assigned integer reference for that interface. `Ipkts` is a count of ingress packets. `Opkts` is a count of egress packets.

```
***   Network Interface Metrics   ***
Name    Index   Ipkts           Opkts
ec0     1       1479223         1843514
lo0     2       7112894         7112894
```

Figure 4: ifchk output under SGI IRIX.

The existence of the ifconfig and netstat programs with their respective abilities to reveal promiscuous mode activity (Figure 2) and display network status data raises a question. What is the use of **ifchk** reporting this information if ifconfig and netstat already do so? The answer to this question lies in trojaned binaries. Attackers will often attempt to conceal their presence on compromised systems so as to allow them to fulfill their objectives, whatever those objectives may be. This sometimes involves the erasure of legitimate ifconfig binaries that will report promiscuous interface activity if the attackers are carrying out any kind of network data reconnaissance. Such binaries are supplanted with compromised versions that will not report promiscuous activity even in the event that an interface is in promiscuous mode.

Netstat, with its ability to display network packet ingress/ egress counts, can also be replaced with a trojaned version. A legitimate netstat binary will increment packet counts by one for every one packet traversing an interface. Consider, however, if a trojaned binary was to increment packet counts by one for every one hundred packets instead.

Recall that a promiscuous interface is not only processing data addressed to it, but a copy of the data addressed to all other systems on the network. This results in a much larger volume of network data to process and, as a result, elevated counts. Such a rise in packet count could alert system administrators to the presence of possible unauthorized activity. The above ingress/egress packet count compromise could, however, aid in concealing such activity.

4. IFCHK IMPLEMENTATION

ifchk utilizes standard operating system services, as supplied by IRIX and Linux, in order to implement its functionality. Examples include ioctl() system calls, the BSD sockets

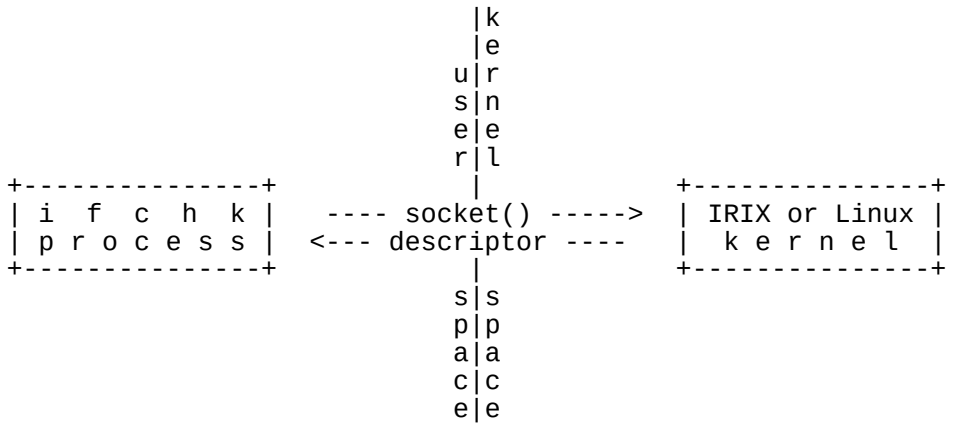

Figure 5: Socket creation prior to flag retrieval.

API and the Netlink/Rtnetlink API. These will be elaborated upon below.

At its core, **ifchk** is concerned with getting and, if told to do so, setting network interface status flags (Figure 2, line 1) which are stored as members of interface specific in-kernel data structures. How this is actually done is operating system specific. IRIX uses ioctl() system calls exclusively while Linux uses a combination of the Netlink/ Rtnetlink API and ioctl(). Both methods will be elaborated upon later.

The manipulation of the data in these flags is a two stage process. First, **ifchk** must establish a communications endpoint, from user space to kernel space, over which interface operations are performed. This is done with the socket() system call (Figure 5) as part of the BSD Sockets API. If successful, the socket() call will return an integer file descriptor referencing our communications endpoint with the kernel.

Under IRIX, **ifchk** then uses the file descriptor returned in the previous step via socket() to send an ioctl() command, as shown in Figure 6. The ioctl() system call provides a facility to control devices such as terminals and and network interfaces via the sending of device specific commands to the kernel. We use ioctl() for the latter category of devices. ioctl() accepts three parameters:

- a socket descriptor referencing a communications endpoint (created above).
- an ioctl() command (described below).
- a device specific data structure, the type of which is a function of the ioctl() command we are sending. The kernel fills in the fields of this data structure that are relevant to the command sent and returns the structure to us.

ifchk sends three different types of ioctl() commands and two different device specific data structures. The command/ data structure pairings are as follows:

`SIOCGIFCONF:` gets a list of all network interfaces present on the system; uses a structure of type ifconf.

`SIOCGIFFLAGS:` retrieves network interface flags from within the kernel; uses a structure of type ifreq.

`SIOCSIFFLAGS:` sets flags on a network interface; uses a structure of type ifreq.

ifconf and ifreq structures are defined in the system header file /usr/include/net/if.h. All **ifchk** IRIX invocations result in at least two ioctl() calls; `SIOCGIFCONF` followed by `SIOCGIFFLAGS`. This produces output in the general format of figure 3 above. `SIOCSIFFLAGS` is only called if a promiscuous interface is to be disabled.

Flag retrieval under Linux begins the same way as under IRIX with the creation of a socket via the socket() system call (Figure 5). Linux then uses a combination of the Netlink/Rtnetlink API and ioctl() to initially get and, if applicable, set interface flags, respectively. ioctl() under Linux will, in certain situations, fail to detect that an interface is in promiscuous mode. Netlink/Rtnetlink cannot be used to disable interfaces. Because of these limitations, an approach using both was required for **ifchk** Linux to satisfy all functional requirements.

Netlink provides us with a method of data transfer between user and kernel space over standard sockets. In creating our socket above, we need to specify what Linux kernel subsystem we wish to communicate with, which, in the case of ifchk, is the `NETLINK_ROUTE` subsystem. `NETLINK_ROUTE` allows us access to Rtnetlink, the Linux-specific implementation of routing sockets. Routing sockets are a method of accessing the in-kernel routing table in order to add or delete routes or to request information about a given route from the kernel. The routing table also contains information on network interfaces including their status flags. ifchk sends one Rtnetlink command, `RTM_GETLINK`,

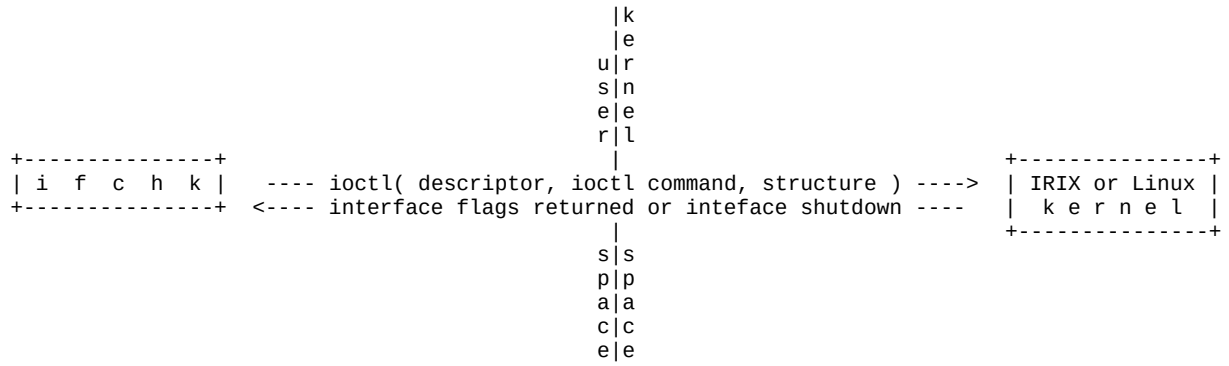

Figure 6: ioctl() command passing.

and uses the data structure that corresponds to that command. The command/data structure pairing is as follows:

RTM_GETLINK: gets information (flags, etc.) about a specific network interface; uses a structure of type ifinfomsg.

The /usr/include/linux/rtnetlink.h system header file defines the ifinfomsg structure. We then call ioctl() to disable the interface if it is in promiscuous mode (as detected by Netlink/Rtnetlink prior) and we are to shut it down.

At its core, **ifchk** uses a large for loop that iterates n times with n being equal to the number of interfaces present on the system as discovered, prior to loop entry, by ioctl() on IRIX or Netlink/Rtnetlink on Linux. For each iteration through the loop, we are testing the flags associated with the current interface under examination and printing its state (Figure 3). In addition to this, we shut that interface down, if applicable. Figure 7 illustrates the main loop.

```
for ( n = firstInterface; n != NULL; n++ )
{
    get interface status;

    if ( promiscuous && disable interface )
    {
        print interface status;
        disable interface;
    }

    else
    {
        print interface status;
    }
}
```

Figure 7: Pseudocode of main program loop.

At loop exit, we perform house cleaning chores such as memory deallocation and the closing of all open file descriptors in conformance with good programming practice.

5. RELATED WORK

Here, we discuss other work related to **ifchk**. CPM (Check Promiscuous Mode) and Sentinel are two such examples of tools that perform promiscuous mode detection.

CPM [1] is a host based promiscuous mode detector that, like **ifchk**, uses ioctl() system calls to print per-interface state information. Written by the Computer Emergency Response Team (CERT), a division of the Software Engineering Institute at Carnegie Mellon University in Pittsburgh, Pennsylvania, CPM reports the number of interfaces present on the system and then, for each interface, prints its name (*e.g.*, `ec0`) and corresponding state. Possible interface states include `Normal` and `*** IN PROMISCUOUS MODE ***`. **ifchk** borrows from this output format as it provides a concise snapshot of interface operation at program runtime. This becomes increasingly important when **ifchk** is run on systems with many interfaces or interface aliases. Tests with **ifchk** Linux on systems with medium to large numbers of interfaces confirm the prudence of this approach.

Sentinel [7] takes a different approach to promiscuous mode detection in that it allows for the detection of promiscuous interfaces on remote systems. Sentinel allows users to perform several different tests to probe for promiscuous mode activity. Examples of tests, as documented by the program author in a previous release, include the following:

- With the DNS test, Sentinel initiates numerous fake TCP connections to nonexistent systems and monitors the network to see if any packet sniffers that might be running on target local systems attempt to resolve, via DNS queries, the IP addresses of those nonexistent systems. Sentinel will then sniff the DNS queries to check to see if it is a target system that is requesting name service resolution.

- With the Etherping test, Sentinel transmits an ICMP echo request message (ping) with a legitimate destination IP address but a fake destination ARP/MAC ad-

dress to a target system. If the target is not sniffing the network, its network hardware will disregard the message. It has been shown, however, that some Linux, NetBSD and Windows NT target systems will, if in promiscuous mode, read such a message and respond to it.

- The Arp test involves Sentinel transmitting an ARP request containing a bogus destination ARP address. A target system not in promiscuous mode would not reply to such a request while a target that was in promiscuous mode would.

ifchk differs from these two tools in that it disables interfaces that it finds in promiscuous mode, thus potentially isolating such systems from the network, and allowing for their subsequent analysis in an isolated environment. **ifchk** also differs from the above two implementations in that it reports per-interface ingress/egress packet counts. This is an aid in the detection of deviations in traffic volume that are at odds with established traffic volume trends.

6. CONCLUSIONS AND FUTURE IFCHK DEVELOPMENT

This paper presented **ifchk**, a method of network interface promiscuous mode detection and interface management. It is a tool that can be used by system and network administrators as an aid in securing systems and networks. It is written in C and made available for free at **http://www.noorg.org/ifchk.** A version for SGI IRIX is currently available, and a Linux version will be released shortly.

There are infrastructural and functional additions planned for future **ifchk** releases with a view toward further developing the program into a serious production level security tool.

1. Divide **ifchk** source code into the following components to facilitate program portability and modularity:

   (a) A framework providing services such as program startup, event logging, user identification and help output.

   (b) An operating system specific module that implements core interface management functionality.

   A module is compiled with the framework to produce an **ifchk** binary. With this design, most of the effort in porting **ifchk** is limited to the creation of new modules.

2. Release a Linux **ifchk** port using the new modularized program architecture as per point 1b above.

3. Turn **ifchk** into a daemon process controlled via an external control utility. The signals facility will be used to provide interprocess control between the daemon and the control utility.

7. ERRATA

This paper has been updated since its initial presentation on May 7th, 2004 at Student and Faculty Research Day hosted by the Pace University School of Computer Science and Information Systems.

REFERENCES

1. **Check Promiscuous Mode**. Open source software available from: ftp://ftp.cerias.purdue.edu/ pub/tools/unix/ sysutils/cpm/.

2. Dhandapani, Gowri, and Sundaresan, Anupama. *Netlink Sockets - Overview*. Available at http://qos.ittc.ukans.edu/ netlink/html/.

3. **ethereal**. Open source software available from: http://www.ethereal.com.

4. Kuznetsov, Alexey. **iproute2**. Open source software available from: ftp.inr.ac.ru/ip-routing.

5. Oualline, Steven. *Practical C Programming, 3rd Edition*. O'Reilly Media Inc, Sebastopol, CA, 1997.

6. Reek, Kenneth A. *Pointers on C*. Addison-Wesley, Boston, MA, 1998.

7. **Sentinel**. Open source software available from: http://www.packetfactory.net/projects/sentinel/.

8. Stevens, W. Richard. *TCP/IP Illustrated, Volume 1: The Protocols*. Addison-Wesley, Boston, MA, 1994.

9. Stevens, W. Richard. *Unix Network Programming, Volume 1, Sockets and XTI, 2nd Edition*. Addison-Wesley, Boston, MA, 1998.

10. **tcpdump**. Open source software available from: http://www.tcpdump.org.