Tracedump: A Novel Single Application IP Packet Sniffer

PAWEŁ FOREMSKI [a]

[a]The Institute of Theoretical and Applied Informatics of the Polish Academy of Sciences
pjf@iitis.pl

Abstract. The article introduces a novel Internet diagnosis utility - an open source IP packet sniffer which captures TCP and UDP packets sent and received by a single Linux process only. Preliminary evaluation results are presented. The utility can be applied in the field of IP traffic classification.

Keywords: Computer networks, traffic monitoring, traffic classification, ptrace, code injection, Linux.

1. Introduction

The Internet needs diagnosis and maintenance tools. One of the most valuable tools for a network administrator is a *packet sniffer*, i.e. a computer program which intercepts IP packets flowing in the network. The output of a sniffer gives insight into how exactly the network operates, and thus can be useful for solving communication issues. For example, in order do diagnose a routing problem, one would observe IP packets on inbound and outbound network interfaces of an Internet router. Presence of network traffic on the inbound interface and absence on the outbound one would mean that the router is not forwarding IP packets properly.

Problem statement

On a particular machine it is difficult to use a packet sniffer in order to capture IP packets belonging to given application only, i.e. packets sent or received by a single process. This is caused by the fact that the most popular sniffers were designed to be deployed on Internet routers, i.e. hosts which rarely generate traffic on their own. A typical packet sniffer can monitor selected network interface, but it lacks enough granularity in order to inspect a local process only. Unfortunately, it seems that such direction in sniffer design influenced operating systems. In particular, the Linux kernel lacks apparatus necessary for straightforward implementation of such functionality in packet sniffers.

The ability of monitoring just a single application is important for several reasons. Original motivation for this work is IP traffic classification using machine learning techniques [1], i.e. development of a computer system which is able to automatically tell the name of application given its IP packets. Before operation, such system needs to be trained using IP traffic trace files annotated with so-called *ground truth* - name of the application that generated these packets. By employing a single application packet sniffer, the problem of *ground truth* is resolved - name of the application is known to the sniffer. In order to obtain an adequate quantity of training data, automation techniques would need to be adopted for practical employment of such method. Synthetic traffic traces are of limited usability, but may be very important during development of traffic analysis and classification systems.

Proposed solution and its scope

This paper presents *tracedump* - a novel IP packet sniffer which intercepts packets belonging to a single application only. It employs several techniques in order to mitigate the lack of necessary mechanisms in the Linux kernel, particularly the *ptrace(2)* [2] system call and the BPF socket filter [3]. The implementation currently supports only TCP and UDP protocols on a 32-bit x86 Linux host, but the proposed approach can be easily applied to different transport protocols, architectures, and possibly to other operating systems.

The sniffer attaches to a given process and to all of its threads and monitors its system calls related to communication with the Internet. A list of local TCP and UDP ports is constructed and used for filtering out all the traffic not belonging to the application under interest.

Paper organization

This paper is organized as follows. Section 2 reviews a few of the most popular packet sniffers available on UNIX-like operating systems, considering their applicability for single application monitoring and for traffic classification. Section 3 gives technical background for the design of *tracedump*. Section 4 presents the internal architecture of the program. Section 5 gives preliminary evaluation results. Section 6 gives a summary, suggesting issues for future work.

2. Related work

One of the most popular packet sniffers is *tcpdump*, accompanied by the *libpcap* library [4]. Originally written in 1987 at the Lawrence Berkeley National Laboratory, it was published a few years later and quickly gained users attention. It runs on most UNIX-like operating systems - e.g. Linux, BSD, Solaris - and on Windows. Since its inception, *tcpdump* was cited by numerous scientific papers in the field of computer networks and is indeed the standard utility for capturing IP traffic. It established an output file format *PCAP*, which is the most popular file format for storing IP packets off-line, still developed to support new functionality [5]. The *tcpdump* sniffer features a command-line filter mechanism, which allows the user to easily capture only the packets satisfying given criteria, e.g. TCP packets with destination port number equal to 80. Unfortunately, this filter mechanism does not support selecting packets of a single application only – especially if the monitored process is a *peer-to-peer* application, allocating new ports each few seconds.

Another very popular packet sniffer, *Wireshark* [6], is a full-fledged GUI application with lots of advanced features. *Libtrace* [7] aims at addressing weaknesses of the *libpcap* library. It supports many input methods and formats, and provides a very good performance. However, none of them can be employed to capture single application traffic.

In the field of traffic classification, there are two notable software utilities. F. Gringoli et al. in [8] present a system for collecting traces of IP traffic, in which flows are annotated with the name of the application that generated them. First, each host in a particular network submits a list of its own connections - along with application names - to the border router of the network. Second, this router captures all of the IP traffic flowing in and out of the network, and by employing the lists submitted by each host, it annotates the resultant traffic trace file with appropriate *ground truth* data. Szabó et al. proposed a similar approach for Windows machines in [9]. Again, none of these two approaches solve the problem of single application diagnosis in a strict sense. They require a separate post-processing stage, what disables possibility of real-time application connectivity diagnosis. They are also prone to loosing IP packets at the beginning of connection, due to non-zero time needed for updating the list of active connections on a particular machine.

3. Design Considerations

The original motivation for *tracedump* was the need to automatically collect samples of network traffic generated by modern desktop applications, as a supportive element for traffic classification systems. Therefore, special care must be taken not to loose any packets, especially those at the beginning of an IP connection. Besides, possibility of capturing all of the DNS queries made by an application may also be crucial for traffic classification purposes.

Let us analyze - in a simplified manner - how a Linux application makes a TCP connection and sends data to a distant host. The operating system provides an API for Internet communication by means of system calls *socket(), connect()*, and *send()*. Thus, the application first calls the *socket()* function in order to get a unique handle for a connection. Then, the *connect()* function is called with the address of the remote peer, and finally the *send()* system call may be used to send the data.

There are two crucial issues one needs to realize when constructing a packet sniffer of a single application. First (A), the application does not handle construction of IP and transport protocol headers - it is the task of the operating system. Hence, it is not enough to intercept the data passed as arguments to system calls responsible for Internet communication. Second (B), a call to *connect()* will generate packets before the call returns. Thus, a packet sniffer must react to *connect()* before it is executed in the kernel.

Unfortunately, it is quite hard to mitigate these problems using existing mechanisms present in the Linux kernel. One of the possible ways to write such a sniffer would be to extend the Linux *struct sk_buff* structure with a *pid* member holding the process ID number. For outgoing packets, this would be trivial, but for incoming packets it could be quite troublesome. However, such approach would constrain the scope of software very much, due to necessity to patch and recompile the operating system kernel.

It is possible to take care of (A) and (B) in user space, without modifying the kernel. A straightforward procedure would be to exploit the dynamic linker *ld.so* [10] in order to provide wrapper functions for system calls responsible for communication with the Internet. However, this would fail for statically compiled program binaries, hence the *tracedump* sniffer implements the *ptrace()* process tracing facility, as will be detailed in the next section.

4. Architecture and Implementation

Tracedump is divided into three functional modules, implemented as threads: *ptrace*, *pcap*, and *garbage collector* (GC). The *ptrace* module attaches to all threads of a given process, and using the Linux *ptrace()* function it constructs a list of all local TCP and UDP ports that the application is using. The *pcap* module operates like an ordinary packet sniffer, intercepting all IP packets on all network interfaces, at the kernel level - recall (A) from the previous section. Whenever the port list changes, a BPF filter is immediately applied on the *pcap* sniffing socket, so that the packets not belonging to the monitored application are ignored. The BPF filter is updated before the kernel executes the original system call - recall (B). The task of the *garbage collector* module is to detect ports that are no longer used. Each minute it reads the list of all active system connections, and it cleans up the list constructed by the *ptrace* thread. The architecture of *tracedump* is depicted on Fig. 4.1.

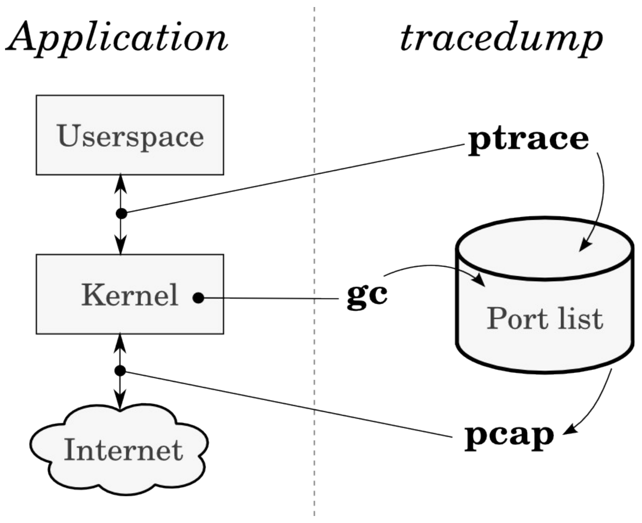

Fig. 4.1: ***Architecture of tracedump.*** *The port list is constructed by observing the kernel-userspace communication and is used for raw IP packet capture. The garbage collector (gc) thread periodically cleans up the list.*

The *ptrace* module traces only three system calls: *bind()*, *connect()*, and *sendto()*. By means of analysis of the Linux kernel source code and by examination of the usual path a user-space program needs to adopt in order to setup an Internet connection, it was verified that - for proposed *tracedump* architecture - this is enough in order not to loose any IP packets. For UDP and TCP servers, the application needs to call *bind()* in order to setup the local port number. For client programs, it will either call *connect()* or *sendto()*. In such case it may happen that the local port number is not yet assigned, and the kernel will perform an “autobind” operation, i.e. allocate an ephemeral port automatically. However, due to (B), this is an undesirable situation, so *tracedump* splits the system call in such case. It forces the process to first call *bind()* with the port argument set to 0, i.e. it requests the automatic allocation to be executed. Then, the BPF filter is updated, and finally the original call - either *connect()* or *sendto()* - is continued. This is realized using machine code injection into the stack area of the monitored process.

The *pcap* thread attaches to the kernel using a *PF_PACKET* [11] socket, and writes captured packets to disk in the PCAP [12] format. Whenever the list of local ports is changed, the BPF filter code is immediately rewritten and sent to the kernel using *setsockopt()* system call.

A naïve solution to tracking local port numbers that the application no longer uses would be to intercept *close()* system calls. Unfortunately, it is not possible to distinguish a *close()* call which effectively ends a connection from a *close()* call which only dissolves an association between a file descriptor and a socket number. The latter may happen in case of multi-threaded applications, which may - or may not - share the file descriptor table amongst its threads. This depends on the detailed configuration of a particular thread, which is difficult to discover on a Linux machine. Thus, *tracedump* utilizes the conventional *procfs* network diagnosis interface, i.e. the */proc/net/tcp* and */proc/net/udp* special files. This interface is quite slow, hence a separate *garbage collector* thread is required in order to continuously read these files in an asynchronous manner.

5. Evaluation

Tracedump has a simple command-line interface. Either a process ID number or a command is accepted as the program argument. The output is a PCAP file, which may be further post-processed with traffic analysis tools like *Wireshark*. It is possible to visually examine the IP packets in real-time, by adopting the UNIX *pipe* mechanism.

On Listing 5.1 below an exemplary application of *tracedump* is presented.

```
1.  root@pjf:~# tracedump ctorrent ubuntu-11.10-alternate-i386.iso.torrent
2.  pcap_init(): Writing packets to dump.pcap
3.  (...)
4.  Total: 673 MB
5.  Creating file "ubuntu-11.10-alternate-i386.iso"
6.  Press 'h' or '?' for help (display/control client options).
7.  | 3/0/754 [1346/1347/1347] 672MB,0MB | 2157,0K/s | 1724,0K E:0,1
8.  Download complete.
```

Listing 5.1: ***Using tracedump to capture BitTorrent traffic.*** *A BitTorrent client ctorrent [13] is used for downloading a CD disk ISO image.*

In this example, an installation CD ISO disk image of a popular Linux distribution is downloaded using the *BitTorrent* [14] protocol. Aim of this experiment is to roughly estimate the overhead of the BitTorrent protocol, in order to present *tracedump*.

In line 1, *tracedump* is started so it monitors a BitTorrent client application downloading a file. In line 2, the resultant PCAP file name is reported: `dump.pcap`. The download process completes in 6 minutes, attaining an average throughput of ~2 MB/s. Table 5.1 presents brief characteristics of the generated IP traffic, obtained using *libtrace* [7] utility programs.

		Packets	Bytes	Ports
TCP	*Outbound*	249,575	17,698,668	77
	Inbound	503,730	718,005,933	
UDP	*Outbound*	2	160	1
	Inbound	2	192	
Total		753,309	735,704,953	78

Table 5.1: ***Characteristics of BitTorrent IP traffic.*** *Last column presents number of transport protocol ports as a sum for inbound and outbound traffic.*

The resultant ISO file is 705,998,848 bytes long, hence a rough overhead of the BitTorrent protocol, including the network and transport protocols, is ~4.21% of the downloaded file size. Note that this also includes all of the DNS queries made during the download process.

6. Conclusion

The article presents *tracedump* - a novel packet sniffer which intercepts IP packets of a single Linux process only. The Linux kernel lacks appropriate mechanisms for straightforward implementation of such utility. Thus, *tracedump* employs several advanced techniques in order to mitigate these deficiencies: *ptrace()* system call, code injection and dynamic generation of BPF assembler code for the Linux socket filter mechanism. The actual process of packet capture happens at the kernel level, so the network and transport protocol headers are retained.

Initial results prove practical applicability of *tracedump*. In the paper, the BitTorrent protocol overhead was roughly estimated by employing *tracedump* to capture IP packets of a BitTorrent client application. Moreover, *tracedump* can be adopted in the field of IP traffic classification. Resultant traffic trace files contain all of the DNS queries made by monitored applications, what opens an interesting prospect for research on employing DNS context as a traffic classification feature. *Tracedump* may also be employed as a general Internet diagnosis utility.

Current implementation of *tracedump* poses a few limits on the scope of practical applications. It works only on 32-bit x86 Linux hosts and is limited to about 300 ports opened at the same time. However, the architecture of *tracedump* leaves plenty of space for future work.

The source code of *tracedump* is published under terms of the GNU General Public License and is available for download from the MuTriCs project website: *http://mutrics.iitis.pl/tracedump*

Acknowledgments

This work was supported by the Polish National Science Centre, under research grant nr 2011/01/N/ST6/07202 - project “MuTriCs” [1].

References

1. MuTriCs: Multilevel Traffic Classification, http://mutrics.iitis.pl/
2. ptrace(2) manual page, http://www.kernel.org/doc/man-pages/online/pages/man2/ptrace.2.html
3. S. McCanne, V. Jacobson, *The BSD packet filter: a new architecture for user-level packet capture*, USENIX Winter 1993 Conference Proceedings (USENIX'93), 1993
4. tcpdump, http://www.tcpdump.org/
5. L. Degioanni, F. Risso, G. Varenni, *PCAP Next Generation Dump File Format*, IETF, Internet-Draft PCAP-DumpFileFormat, 2004
6. Wireshark, http://www.wireshark.org/
7. S. Alcock, P. Lorier, R. Nelson, *Libtrace: A Packet Capture and Analysis Library*
8. F. Gringoli, L. Salgarelli, M. Dusi, N. Cascarano, F. Risso, K.C. Claffy, *GT: picking up the truth from the ground for Internet traffic*, ACM SIGCOMM Computer Communication Review, Vol. 39, No. 5, pp. 13-18, Oct. 2009
9. G. Szabó, D. Orincsay, S. Malomsoky, I. Szabó, *On the validation of traffic classification algorithms*, Proceedings of PAM'08, Springer-Verlag, 2008
10. ld.so(8) manual page, http://www.kernel.org/doc/man-pages/online/pages/man8/ld.so.8.html
11. packet(7) manual page, http://www.kernel.org/doc/man-pages/online/pages/man7/packet.7.html
12. PCAP file format, http://wiki.wireshark.org/Development/LibpcapFileFormat
13. ctorrent, http://ctorrent.sourceforge.net/
14. B. Cohen, *The BitTorrent Protocol Specification*, http://bittorrent.org/beps/bep_0003.html