
ERNW Newsletter 39 / March 2012

Attacking High Speed Ethernet Links - Practical attacks against unencrypted high speed Ethernet links.

Version: 1.0

Date: 12.03.2012

Author: Enno Rey (erey@ernw.de), Daniel Mende (dmende@ernw.de), Hendrik Schmidt (hschmidt@ernw.de), Matthias Luft (mluft@ernw.de)

ERNW Enno Rey Netzwerke GmbH

Breslauer Str. 28

69124 Heidelberg

Tel. +49 6221 480390

Fax +49 6221 419008

www.ernw.de

Table of Contents

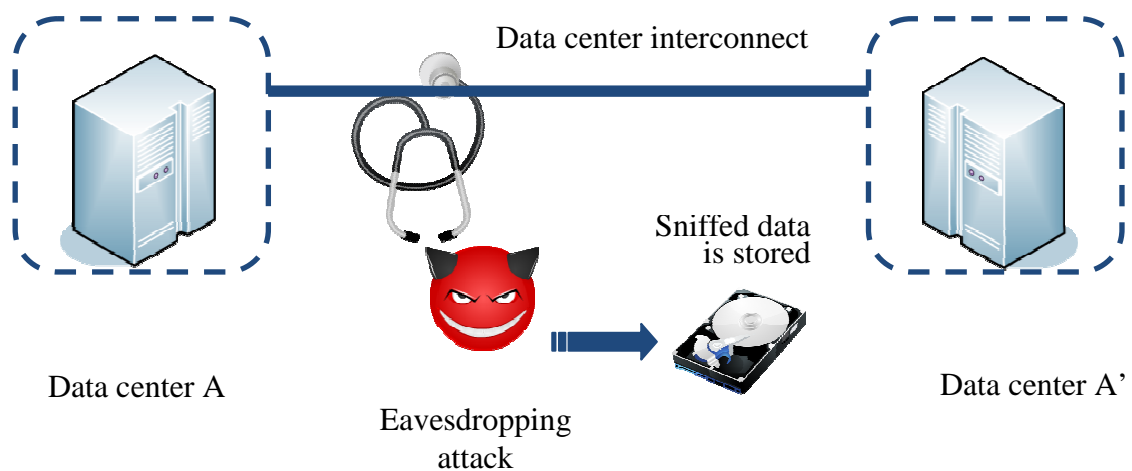
1	ABSTRACT	3
2	EXECUTIVE SUMMARY	3
3	INTRODUCTION	5
3.1	Problem Statement	5
3.2	Related Work	5
3.3	Prerequisites	5
4	USE CASE: DATA CENTER INTERCONNECT	6
4.1	Involved Network Technologies	6
4.2	Protocols	6
4.3	Attacks.....	6
5	TACKLING THE PROBLEM	7
5.1	COTS Packet Analysis Tools	7
5.2	Custom Tool pcap_extractor	7
5.3	Hardware Aspects	7
5.3.1	Identifying the bottleneck	7
5.3.2	Actual lab setup	8
5.4	Doing it “the cloud way”	8
5.4.1	Lab Setup	8
5.4.2	Results	9
6	TEST RESULTS	10
6.1	Description of Overall Test Methodology	10
6.1.1	Command line options	10
6.2	Results	11
6.3	Demo: Extracting credit card data from an Oracle database running on virtual machine being transferred as a whole	13
7	CONCLUSIONS	15
8	APPENDIX	16
8.1	Bibliography	16
8.2	Details of AWS Instance used	17
8.3	Code of custom tool.....	18

1 ABSTRACT

This paper discusses practical attacks against unencrypted high speed Ethernet links. There is a common misconception that the sheer amounts of data which can be transferred using multiplexed channels (e.g. WDM technology) make successful attacks highly unlikely. We will show that a skilled and motivated attacker can easily identify and extract sensitive information if he observes/collects a large amount of raw data and, more importantly, will be able to do so in a feasible manner as for the time or technical resources needed.

2 EXECUTIVE SUMMARY

To assess if the extraction of specific data from very large sets of captured network traffic can be done in a feasible way in terms of tools, hardware or time needed, a practical approach was undertaken. This included evaluating tools, building a lab with suitable hardware for performing a number of tests, including a demonstration of extracting credit card data from a file sized 500 GB and containing a virtual machine running an *Oracle* database. Such a file could, for example, have been derived from an eavesdropping attack against a high speed link connecting two data centers of an organization.

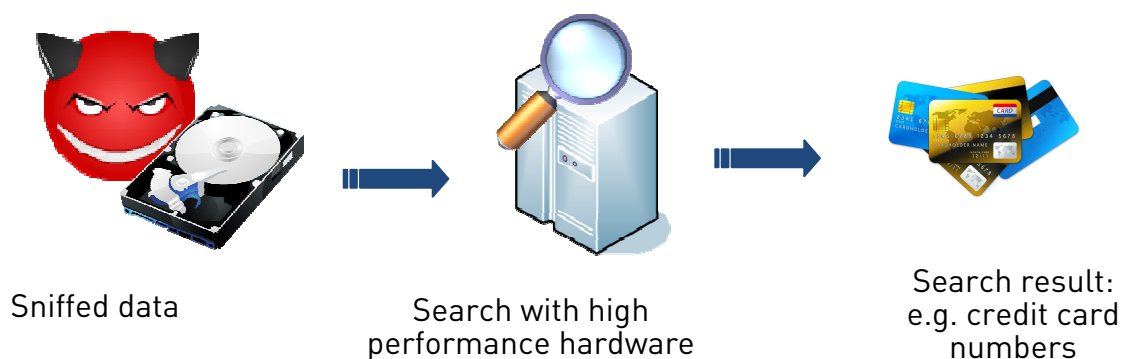


Assuming the captured data was stored in standard *pcap* format on a number of equally-sized solid-state drives (SSDs) the data extraction task was proven quite feasible with somewhat limited resources. Table 1 shows the most important results.

Table 1: Results from test lab configured with four individual SSD drives and using custom tool

Filesize [GB]	Time of custom code [s]	Estimated time for 500 GB [s]	Time of tcpdump [s]	Estimated time for 500 GB [s]
44	113	1284	114	1295
80	203	1269	204	1275
204	521	1277	522	1279
404	1017	1259	1016	1257
516	1290	1250	1290	1250
808	2041	1263	2043	1264

As depicted, in the demonstration setup, the search for particular credit card data could be performed in about 21 minutes, employing available tools and using common-of-the-shelf hardware for about 3000 €.



Furthermore the time needed scales linearly with the file size, so that processing a 1 TB data volume presumably would have taken ~ 42 minutes, a 2 TB file would have taken ~ 84 minutes and so on. In addition, SSD prices are constantly declining, too.

Thus it could be shown that simply the amount of data gained from eavesdropping high speed links may not prohibit an attacker from analyzing this data.

3 INTRODUCTION

3.1 Problem Statement

There is a common misconception that the sheer amounts of data coupled with multiplexed channels (e.g. WDM technology) make successful eavesdropping attacks on high speed Ethernet¹ links highly unlikely. This is mainly based on the assumption that the amount of resources (e.g. RAM, [sufficiently fast] storage or CPU power) needed to process large files of captured data is a limiting factor. However, to the best of our knowledge, no practical evaluation of these assumptions has so far been performed.

Therefore this paper aims to answer the following questions:

- Can large² amounts of captured data be processed “in a feasible way”³?
- How much time and which type of hardware is needed to perform this task?
- Can this be done with publicly available tools or is custom code helpful or even required? If so, how should that code operate?
- Can this task be facilitated by means of public cloud services?

We performed a number of tests with files of different sizes and entropy⁴. Tests were both carried out with different sets of dedicated hardware and by means of public cloud services. This paper describes the tools used, the various test setups and, of course, the results. A final section includes some conclusions derived from the insights provided by the test sets.

While our research wasn’t focused on particular environments, the analysis of typical “data center interconnect links” will be discussed. For that purpose a lab was built where a full transfer of an *Oracle* database running on a virtual machine (being transferred across such a link by means of *VMotion* technology) and processing credit card data was simulated.

3.2 Related Work

We did not perform an extensive search for *related work* but it seems a widely accepted fact that working with large pcap files might become cumbersome and often leads to systems crashing when those are opened with *Wireshark*. Even though there are several papers discussing the performance of network analysis [1] or the distribution of this analysis[2], none of those covers the problem discussed in this paper.

3.3 Prerequisites

It is assumed that an attacker has already gained access enabling him to eavesdrop on the high speed data link. A detailed description how this can be done can be found in [3]. The focus of the present paper is on the subsequent extraction of useful data from the resulting dump file. It is further assumed the collected data is available in standard pcap⁵ format.

¹ In the context of this paper “high speed Ethernet” refers 10 GbE or faster.

² For demonstration purposes we limited our research to files in the 500 gigabyte size range (with 500 gigabyte being equivalent to the full live migration of 16 virtual machines with 32 GB of virtual memory each) which equates to ~ 400 seconds of sniffing on a full 10 GbE link. However it can be shown (and will be laid out in detail in this paper) that the analysis time of traffic dump files scales up almost linearly. So, in the end of the day, it comes down to the hardware of (more precisely: the type and number of the storage devices attached to) the system performing the actual processing.

³ Where “feasible” means within some hours and by means of hardware to be purchased for less than 5,000 EUR.

⁴ Entropy is defined here as “a measure of disorder, or [...] unpredictability” (from Wikipedia).

⁵ Pcap API - Libcap (Unix systems), www.tcpdump.org.

4 USE CASE: DATA CENTER INTERCONNECT

Many organizations dispose of redundant data centers, be it for high availability or disaster recovery purposes, be it (in some countries and industry branches, namely in finance/banking) for compliance reasons. These are often located in close proximity (within the 10–50 km range) to each other and are then connected by means of high speed Ethernet links allowing for continuous, near real-time data mirroring, for SAN replication or for fast service failover including the transfer of virtual machines by technologies like VMware's *VMotion*.

4.1 Involved Network Technologies

There are three main approaches for the network interconnection between such data centers:

- Dedicated dark fiber, used by just one customer/organization.
- Dedicated lambda on a WDM link.
- Use of some *Carrier Ethernet* technology which encompasses Metro Ethernet, XoMPLS, VPLS or Ethernet over SDH (EoSDH).

It should be noted that none of these dispose of inherent security properties (like traffic authentication or confidentiality/integrity protection or the like) by design. Their – potential or real – security is solely based on the perceived isolation property they provide and the expected trustworthiness of the environment they run in, e.g. a carrier's network or premises.

4.2 Protocols

The most common protocols for SAN traffic or replication are NFS, FCoE and iSCSI. While a detailed discussion of their respective functionality and specifications is not relevant for our discussion, it should be noted that, again, most of them do not dispose of inherent security properties on their own. A notable exception is NFSv4 which has some mature security mechanisms. However, the authors are not aware of many organizations using these mechanisms.

4.3 Attacks

Given the absence of any security mechanisms on these links (neither on the network layer nor on the protocol/application layers) all types of attacks against the transported traffic's confidentiality or integrity might be possible – once an attacker gets access to a link⁶.

These include eavesdropping attacks against data stored in data bases (like credit card information from customers as shown in the sample attack below) as well as more sophisticated attacks against authentication protocols.

In 2008, an American PhD student demonstrated how to effectively circumvent the SSH authentication of systems transferred by means of *VMotion*. A single instruction which was injected into the runtime image of the SSH daemon on a live machine⁷ enabled the attacker to circumvent the SSH authentication mechanisms.

This shows that successful attacks against such links might have disastrous consequences. Complementing the high impact of a possible attack, it must be shown that it also can be performed in a feasible way, so to speak in a reasonable amount of time and using commodity hardware.

⁶ See (3).

⁷ See <http://www.blackhat.com/presentations/bh-dc-08/Oberheide/Presentation/bh-dc-08-oberheide.pdf> for details. It should be noted that with vSphere 4.0 a parameter was introduced that allows to require the encryption of VMotion transfers but first this one has some usability problems on its own (see <http://virtualkenneth.com/2009/08/11/v/>) and second from the authors' assessment experience is rarely used in real-world deployments.

5 TACKLING THE PROBLEM

This section outlines pitfalls of the analysis of very large traffic dump files that were identified during the initial research phase or are based on experiences from the actual lab setup and test runs. Based on these findings, it was possible to build the final lab setup in an efficient way.

5.1 COTS Packet Analysis Tools

A number of tests utilizing available command-line tools (tetherreal, tshark, tcpdump⁸ and the like) were performed. It turned out that, performance-wise, "classic" tcpdump showed the most promising results. During the following, in-depth testing phase two problems of tcpdump showed up:

- As it is single-threaded, it cannot use multiple processors of a system (for parallel processing). However, given the actual bottlenecks to be related with I/O anyway (see below) this issue turned out not to be a major problem.
- Standard pcap filters do not allow for "keyword search" in an easy way⁹ which somehow limits the attack scenarios (attacker might not be able to search for credit card numbers, user names etc. but would have to perform an IP parameter based search first and then hand over to another tool which might cause an unacceptable delay in the overall analysing process).

5.2 Custom Tool pcap_extractor

The custom tool developed in the course of the research process is basically the fastest possible implementation of a pcap file reader. It opens a libpcap file handle for the designated input file, applies a libpcap filter to it and loops through all the filter matching packets, writing them to an output pcap file. Contrary to tcpdump and most other libpcap based analysis tools, it provides the possibility to search for a given string inside of the matching packets, for example a credit card number or a username in an easy way. If such a search string is applied, only packets matching the libpcap filter and containing the search string are written to the output file.

An example call to search a pcap file for iSCSI packets which contain a certain credit card number and write them to the output file is shown in the following:

```
$> pcap_extractor -i input-file.pcap -o output-file.pcap -f "tcp port 3260" -s "5486123456789012"
```

5.3 Hardware Aspects

5.3.1 Identifying the bottleneck

While measuring the performance of multiple pcap analysis tools, profiling of system calls indicated that the tools spent between 85% and 98%¹⁰ of the search time on waiting for I/O. In case of the fastest tool that means 98% of the time the tool doesn't process anything, but waits for dump data. So the I/O bandwidth turned out to be the major bottleneck in the test setup.

⁸ <http://www.tcpdump.org/>.

⁹ It is possible to split up the search string into byte numbers and insert into the pcap filter, but this is quite difficult for using the tool in a fast and easy way. Additionally, in case of using tcpdump, we found out that there are some problems with NULL-bytes inside a packet.

¹⁰ Depending on the tool analyzed.

5.3.2 Actual lab setup

The lab system was designed to provide as much I/O bandwidth as possible and was composed of:

- Intel Core i7-990X Extreme Edition, 6x 3.46GHz
- 12GB (3 * 4GB) DDR3 1600MHz, PC3-12800
- ASRock X58 Extreme6 S1366 mainboard
- 4 * Intel 510 Serie Elm Crest SSD 250GB

The mainboard and the SSDs were chosen to support SATA3 with a theoretical maximal I/O bandwidth of 6 Gbit/s. FreeBSD was used as operating system.

5.4 Doing it “the cloud way”

One of the key characteristics of cloud computing is the flexible provision of computing resources on demand. The processing of large data sets on peak loads is a typical use case for cloud computing environments. Therefore, the I/O intensive analysis of the large pcap files was also performed within the *Amazon Elastic Compute Cloud* where a large amount of resources could be allocated as they were necessary. Additionally, virtualized cloud storage is quickly and available in arbitrary size.

5.4.1 Lab Setup

The *Amazon Elastic Compute Cloud* (short: EC2) provides a flexible environment for the provisioning of virtual machines of different hardware performance on demand. For the current lab setup, an *extra large instance* (see Appendix 8.2) was used. Since the I/O performance of a single disk was the bottleneck of the data processing, eight *Elastic Block Storage* (short: EBS) volumes were created and attached to the instance. Each EBS volume is hosted within a specific *availability zone* and can be attached to instances running in the same zone. EBS volumes can be created and attached issuing two commands of the amazon ec2 command line tools. Therefore the amount of storage can be scaled up very easily. The only pre-condition is the existence of a sufficient number of EBS volumes which contain parts of the pcap file to be analyzed. During the benchmarks, the performance was significantly lower than the one of the system presented in chapter 5.3.2, even though eight different EBS volumes were used to avoid the bottleneck of a single storage volume. The overall performance of the test was limited by the I/O performance restriction within virtualized instances and virtualized storage systems. Following the overall cloud computing paradigm, performance limitations of this kind can be circumvented by using multiple resources which do the processing in parallel. This can be done by using multiple instances or by using frameworks like Amazon *MapReduce* which are designed to process huge sets of data. Applying this approach to the analysis of pcap files, the structure of the pcap format carries some inherent problems. This format consists of a binary representation of the data which is structured by the time of the captured packets and not by logical packet traces. Therefore it would be necessary to process the complete pcap file by each instance to extract all streams to identify which streams of the file are to be analyzed by the concrete worker instance. This prevents an efficient distribution of the analysis in multiple jobs or input files. If the captured network data would be stored in separate streams instead of one big pcap file, the processing using a map/reduce algorithm would be possible and thus significantly increase scalability.

5.4.2 Results

Table 2 shows the results of the analysis in the cloud environment described in Section 5.4.1. The evaluated costs include storage and computing power, additional I/O cycles or input bandwidth was not considered.

Table 2: Results of the cloud setup

Filesize (GB)	Time of custom code (s)	Estimated time based on 500 GB (s)	Costs (\$)
50	380	424	0.07
100	827	848	0.14
204	-	1713	0.36
404	-	3393	1.17
500	4242	-	1.63
808	-	6787	3.39

6 TEST RESULTS

6.1 Description of Overall Test Methodology

To evaluate the performance of the testing environments which were used to analyze capture data, two different tools were used. At first, the still state-of-the-art network analysis tool *tcpdump*, a simple and powerful packet analyzer, was used. At second, a custom tool¹¹, called *pcap_extractor*, was specifically developed. The custom tool is similar to the *tcpdump* utility in the sense both work with the *libpcap* C/C++ library. Both tools support packet filtering by means of *libpcap* filter expressions. Additionally the custom tool also supports searching for strings inside of a network packet. For the tests, five capture files were created using the *mergcap* utility. Different sample traffic dumps¹² were merged to five large files with different file sizes. All these files consist of several capture files containing a variety of protocols (as well iSCSI and FCoE). In this context, capture files of ~40, ~80, ~200, ~500, and ~800 Gigabytes were created and were analyzed with both tools. At all tests the filtering expressions for *tcpdump* and *pcap_extractor* were configured to search for a specific source IP address and a specific destination IP address matching for some iSCSI packets inside of the capture file.

To address the performance bottleneck of the tools, which had been identified to be the I/O throughput, two different testing environments (see above) were implemented. On the one hand, the lab was configured with *raid0* storage technology, using four SSD hard drives, and on the other hand, four SSDs were used as single drives with each of them processing only a fourth of the analyzed capture file. For splitting up the capture files into smaller ones, tools like *dumpcap*, *tcpdump* or *splitcap* and *editcap* (which are part of the *Wireshark* suite) can be used¹³.

The UNIX program *time* was used to measure the time of execution. Additionally the tools analyzing the data were started with the highest possible scheduling priority¹⁴ to ensure execution with the maximum of available resources.

6.1.1 Command line options

For the custom tool:

```
$> /usr/bin/time -hp /usr/bin/nice -n -19 ./pcap_extractor -i in.pcap -o out.pcap -f "ip src 192.168.1.207 and ip dst 192.168.1.208" > out &
```

For the custom tool with string search:

```
$> /usr/bin/time -hp /usr/bin/nice -n -19 ./extractor -i in.pcap -o out.pcap -f "ip src 192.168.1.207 and ip dst 192.168.1.208" -s "0000000000620012" > out &
```

For *tcpdump*:

```
$> /usr/bin/time -hp /usr/bin/nice -n -19 /usr/sbin/tcpdump -r in.pcap -w out.pcap src 192.168.1.207 and dst 192.168.1.208 > out &
```

¹¹ See section 5.2 or appendix 8.4 for a more detailed description.

¹² <http://wiki.wireshark.org/SampleCaptures>

¹³ Obviously this might already be done as part of the capturing/collection process to save time later on.

¹⁴ This was done using the Unix *nice* command.

6.2 Results

In the following the lab results are provided and discussed.

Table 3 shows the times for the tools using the raid0 storage technology. To get a better relation between the different file sizes, an interpolated time, based on the time of the current file size, for 500 gigabytes is depicted in addition. This time gives the possibility to compare all the times of different file sizes. The result using the raid0 technology gives a first overview of the programs' runtimes. While in almost all test cases the custom code seems a little bit faster than tcpdump, the estimated time for 500 GB is around 2400 seconds (40 minutes).

Table 3: Configured with raid0

Filesize (GB)	Time of custom code (s)	Estimated time for 500 GB (s)	Time of tcpdump (s)	Estimated time for 500 GB (s)
44	230	2614	230	2614
80	386	2413	387	2419
204	1003	2458	1006	2466
404	1932	2391	1938	2399
516	2477	2400	2487	2410
808	3877	2399	3883	2403

Table 4 shows the second test setup using 4 single SSD drives which all contain a fourth of the capture file to be analyzed. In comparison to the first scenario, the analysis time for the 500 GB file decreases significantly (about 21 minutes).

Table 4: Configured with 4 single SSD drives

Filesize (GB)	Time of custom code (s)	Estimated time for 500 GB (s)	Time of tcpdump (s)	Estimated time for 500 GB (s)
44	113	1284	114	1295
80	203	1269	204	1275
204	521	1277	522	1279
404	1017	1259	1016	1257
516	1290	1250	1290	1250
808	2041	1263	2043	1264

Table 5 shows the results of an additional test. During this test, besides the normal filtering expression, the program is also searching for a specific string inside of the network packets. Looking at the results it becomes clear that there is no relevant deviance to the runtime of the program without searching for a specific string.

Table 5: Times for searching a string with pcap_extractor

Filesize [GB]	Single SSD drives , time [s]	Estimated time for 500 GB [s]	Raid 0, time [s]	Estimated time for 500 GB [s]
44	113	1284	229	2602
80	203	1269	384	2400
204	517	1267	1002	2456
404	1010	1250	1930	2389
516	1288	1248	2476	2399
808	2022	1251	3874	2397

Figure 1 displays the runtimes of both laboratory configurations. On the one hand it is possible to derive a comparison between the raid0 and the single drive configuration and on the other hand the figure allows for an estimation as for other file sizes not used in the tests.

All tests lead to the assumption that there is linearity towards larger filesizes and there is a possible speed-up for a faster processing via configuring the lab with more parallelized I/O capacity.

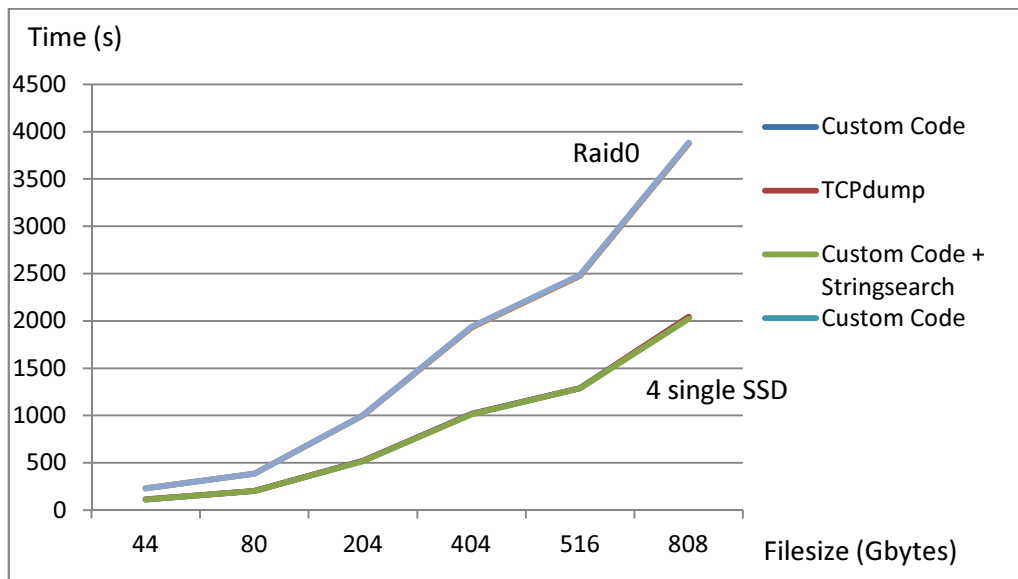
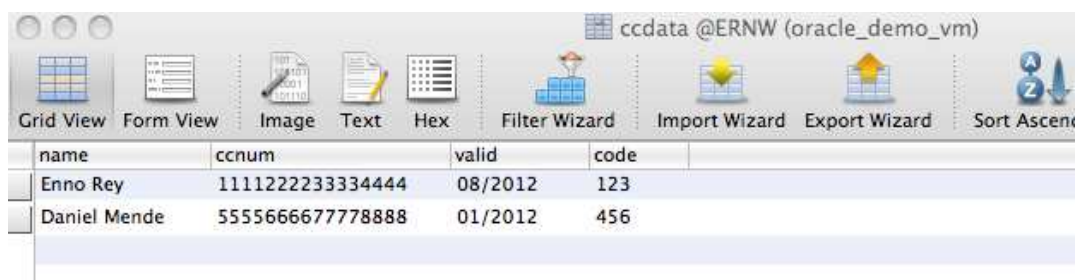


Figure 1: Linearity of the work process

6.3 Demo: Extracting credit card data from an Oracle database running on virtual machine being transferred as a whole

To show the impact of capturing and filtering data in business environments, an oracle database was set up on a virtual machine. Such an Oracle database may store a lot of information, which in addition may be send over the network due to various reasons. There may be a database transfer between two locations, a backup to another server or just a simple transaction by a user. Figure 2 shows the content of the Oracle database, set up for simulating an environment storing sensible credit card data.



name	ccnum	valid	code
Enno Rey	1111222233334444	08/2012	123
Daniel Mende	5555666677778888	01/2012	456

Figure 2: Contents of the Oracle database

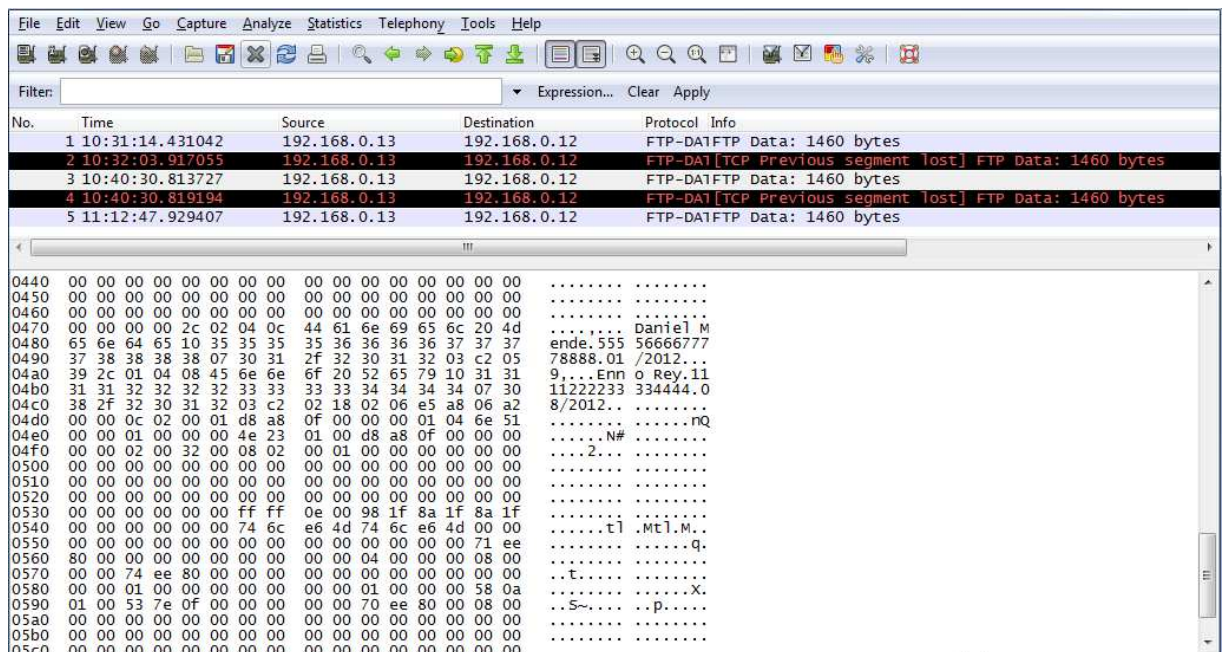
At first we started a transfer of the whole virtual machine which is running the Oracle database. For reasons of simplicity, FTP was used instead of VMotion. By default both approaches do not encrypt the transferred data.

This capture file was then analyzed with `pcap_extractor` to search for a specific string (for example the name of a credit card holder), which is shown in Figure 3.

```
droelf# ./pcap_extractor -i 0/vm_transfer.pcap -o out_ennorey.pcap -f "host 192.168.0.13" -s "Enno Rey"
pcap highspeed extractor version 1      by Daniel Mende - dmende@ernw.de
Found 5 matching packets in 103 seconds.
droelf#
```

Figure 3: Console output of `pcap_extractor`

The resulting output file contains the packets with the data searched for, as shown in Figure 4.



No.	Time	Source	Destination	Protocol	Info
1	10:31:14.431042	192.168.0.13	192.168.0.12	FTP-DAI	FTP Data: 1460 bytes
2	10:32:03.917055	192.168.0.13	192.168.0.12	FTP-DAI	[TCP Previous segment lost] FTP Data: 1460 bytes
3	10:40:30.813727	192.168.0.13	192.168.0.12	FTP-DAI	FTP Data: 1460 bytes
4	10:40:30.819194	192.168.0.13	192.168.0.12	FTP-DAI	[TCP Previous segment lost] FTP Data: 1460 bytes
5	11:12:47.929407	192.168.0.13	192.168.0.12	FTP-DAI	FTP Data: 1460 bytes

0440	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
0450	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
0460	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
0470	00 00 00 00 00 2c 04 0c	44 61 6e 69 65 6c 20 4d Daniel M
0480	65 6e 64 65 10 35 35 35	35 36 36 36 36 37 37 37	ende.555 56666777
0490	37 38 38 38 07 30 31	2f 32 30 31 32 03 c2 05	78888.01 /2012...
04a0	39 2c 01 04 08 45 6e 6e	6f 20 52 65 79 10 31 31	9,...Enn o Rey.11
04b0	31 31 32 32 32 33 33	33 33 34 34 34 34 07 30	11222233 334444.0
04c0	38 2f 32 30 31 32 03 c2	02 18 02 06 e5 a8 06 a2	8/2012... ..
04d0	00 00 0c 02 00 01 d8 a8	0f 00 00 00 01 04 6e 51nQ
04e0	00 00 01 00 00 00 4e 23	01 00 d8 a8 0f 00 00 00N#
04f0	00 00 02 00 32 00 08 02	00 01 00 00 00 00 00 002...
0500	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
0510	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
0520	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
0530	00 00 00 00 00 ff ff	0e 00 98 1f 8a 1f 8a 1f
0540	00 00 00 00 00 74 6c	e6 4d 74 6c e6 4d 00 00t] .Mt]M..
0550	00 00 00 00 00 00 00 00	00 00 00 00 00 00 71 ee
0560	80 00 00 00 00 00 00 00	00 00 04 00 00 00 08 00q.
0570	00 00 74 ee 80 00 00 00	00 00 00 00 00 00 00 00T.....
0580	00 00 01 00 00 00 00 00	00 00 01 00 00 00 58 0aX.
0590	01 00 53 7e 0f 00 00 00	00 00 70 ee 80 00 08 00S~...p.....
05a0	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
05b0	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
05c0	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00

Figure 4: Output file of pcap_extractor in Wireshark

7 CONCLUSIONS

It could be shown that the extraction of specific data from very large network traffic dumps can be achieved within a rather short time period. This was even possible using COTS hardware available for about 3000 € (as of March 2011). This means that an attacker disposing of (large) data sets resulting from previous eavesdropping attacks will most likely succeed in getting the exact data she's going after. In the lab setup, it took about 21 minutes to find a certain credit card number within a file sized 500 gigabyte which equates to a live migration of 16 virtual machines with 32 GB of virtual memory each.

The widespread perception that the sheer amount of data transferred over high speed network links prevents the extraction of data could thereby be proven wrong.

The entropy of a given file has practically no influence on the search/processing time needed. Furthermore it should be noted that the tests show a high correlation/degree of linearity between the size of the files (potentially derived by splitting one very large file into several chunks, each of them still of huge size) and the search time. So it seems there's a simple trade-off between the amount of storage provided and the search time. Given the ever-declining price of storage this will lead to even lower costs for such an attack in the future.

8 APPENDIX

8.1 Bibliography

1. **Nottinghamn, Alastair and Irwin, Barry.** *Parallel packet classification using GPU co-processors*. Rhodes : s.n., 2010.
2. **Kang, Wonchul, Lee, Yeonhee and Lee, Youngseok.** *Netflow Analysis with MapReduce*. Chungnam : s.n., 2010.
3. **Infoguard.** Risiken und Gefahren bei optischen Datenleitungen. [Online] [Zitat vom: 15. 5 2011.] http://www.infoguard.com/docs/PDF/IG_Dokumente/WP_Fiber_Optic_Communication-d.pdf.
4. **Konwinski, Andy and Rabkin, Ariel.** *Using multi-source data to analyze large packet traces*. Berkeley : s.n.
5. **But, Jason and Bussiere, Julie-Anne.** *Improving NetSniff Capture Performance on FreeBSD by Increasing the PCAP Capture Buffer*. Melbourne : s.n., 2005.
6. **Vijn, Arin.** *A 10GE Monitoring System*.

8.2 Details of AWS Instance used

The used Amazon EC2 instance was a so called *extra large instance*. Amazon describes this instance type as follows:

- 15 GB memory
- 8 EC2 Compute Units (4 virtual cores with 2 EC2 Compute Units each)
- 1,690 GB instance storage
- 64-bit platform
- I/O Performance: High
- API name: m1.xlarge

1 EC2 Compute Unit “provides the equivalent CPU capacity of a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor”¹⁵, whereby the I/O performance scale of low, medium and high is not defined in more detail.

¹⁵ As stated under <http://aws.amazon.com/ec2/instance-types/>.

8.3 Code of custom tool

```
//      pcap_analyser.c
//
//      Copyright 2011 Daniel Mende <dmende@ernw.de>
//
//      This program is free software; you can redistribute it and/or modify
//      it under the terms of the GNU General Public License as published by
//      the Free Software Foundation; either version 2 of the License, or
//      (at your option) any later version.
//
//      This program is distributed in the hope that it will be useful,
//      but WITHOUT ANY WARRANTY; without even the implied warranty of
//      MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
//      GNU General Public License for more details.
//
//      You should have received a copy of the GNU General Public License
//      along with this program; if not, write to the Free Software
//      Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston,
//      MA 02110-1301, USA.

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/time.h>

#include <pcap.h>

#define MAX_PACKET_LEN 4096

void print_help()
{
    printf("\n-h\n");
    printf("-i <infile>\n");
    printf("-o <outfile>\n");
    printf("-f <pcap-filter>\n");
    printf("-s <search-string>\n\n");
    exit(0);
}

int main(int argc, char **argv)
{
    int opt;
    char *infile = NULL;
    char *outfile = NULL;
    char *filter = NULL;
    char *search = NULL;

    pcap_t *pcap_handle_in;
    char pcap_errbuf[PCAP_ERRBUF_SIZE];
    const u_char *pcap_packet;
    struct pcap_pkthdr *pcap_header;
    struct bpf_program pcap_filter;
    pcap_dumper_t *pcap_dumper;
```

```

char packet[MAX_PACKET_LEN + 1];
int len;

struct timeval start, stop;
unsigned long found = 0;

printf("pcap highspeed extractor version 1\tby Daniel Mende - dmende@ernw.de\n");
fflush(stdout);

while ((opt = getopt(argc, argv, "i:o:f:s:h")) != -1) {
    switch (opt) {
        case 'i':
            infile = optarg;
            break;
        case 'o':
            outfile = optarg;
            break;
        case 'f':
            filter = optarg;
            break;
        case 's':
            search = optarg;
            break;
        case 'h':
            print_help();
            break;
        default:
            printf("unknown switch: %c\n", opt);
            return 1;
    }
}

if (!infile || !outfile || !filter) {
    printf("oops, missing some argument...\n");
    print_help();
    return 1;
}

// open infile
pcap_handle_in = pcap_open_offline(infile, pcap_errbuf);
if (pcap_handle_in == NULL) {
    fprintf(stderr, "Couldn't open file: %s\n", pcap_errbuf);
    return 2;
}
if (pcap_compile(pcap_handle_in, &pcap_filter, filter, 0, 0) == -1) {
    fprintf(stderr, "Couldn't parse filter: %s\n", pcap_errbuf);
    return 2;
}
if (pcap_setfilter(pcap_handle_in, &pcap_filter) == -1) {
    fprintf(stderr, "Couldn't install filter: %s\n", pcap_errbuf);
    return 2;
}

// open outfile
pcap_dumper = pcap_dump_open(pcap_handle_in, outfile);
if (pcap_dumper == NULL) {
    fprintf(stderr, "Couldn't open file: %s\n", pcap_errbuf);
    return 2;
}

```

```

    }

    gettimeofday(&start, NULL);
    while (pcap_next_ex(pcap_handle_in, &pcap_header, &pcap_packet) > 0) {
        if (search) {
            if (pcap_header->len > MAX_PACKET_LEN) {
                memcpy(packet, pcap_packet, MAX_PACKET_LEN);
                packet[MAX_PACKET_LEN] = '\0';
                len = MAX_PACKET_LEN;
            } else {
                memcpy(packet, pcap_packet, pcap_header->len);
                packet[pcap_header->len] = '\0';
                len = pcap_header->len;
            }
            if (memmem(packet, len, search, strlen(search))) {
                found++;
                pcap_dump((u_char *) pcap_dumper, pcap_header, pcap_packet);
            }
        } else {
            found++;
            pcap_dump((u_char *) pcap_dumper, pcap_header, pcap_packet);
        }
    }
    gettimeofday(&stop, NULL);

    pcap_dump_close(pcap_dumper);
    pcap_close(pcap_handle_in);

    printf("Found %lu matching packets in %i seconds.\n", found, (int) (stop.tv_sec -
start.tv_sec));

    return 0;
}

```