Network Data Capture in Honeynets

Berkeley Packet Capture (BPF) and Related Technologies : An Introduction

Alexandre Dulaunoy

a@foo.be

November 20, 2014

Promiscuous mode

Where can we capture the network data ? a layered approach

- *A network card can work in two modes, in non-promiscuous mode or in promiscuous mode :*
  - *In non-promiscuous mode, the network card only accept the frame targeted with is own MAC or broadcasted.*
  - *In promiscuous mode, the network card accept all the frame from the wire. This permits to capture every packets.*

```
ifconfig eth0 promisc
```

- *Other approaches possible to capture data (Bridge interception, dup-to of a packet filtering, ...)*

A side note regarding wireless network, promiscuous mode is only capturing packet for the associated AP. You'll need the monitor mode, to get capturing everything without being associated to an AP or in ad-hoc mode.

BPF History

How to get the data from the data link layers ?

- *BPF (Berkeley Packet Filter) sits between link-level driver and the user space. BPF is protocol independant and use a filter-before-buffering approach. (NIT on SunOS is using the opposite approach).*
- *BPF includes a machine abstraction to make the filtering (quite) efficient.*
- *BPF was part of the BSD4.4 but libpcap provide a portable BPF for various operating systems.*
- *The main application using libpcap (BPF) is tcpdump. Alternative exists to libpcap from wiretap library or Fairly Fast Packet Filter.*

Network data capture is a key component of a honeynet design.

BPF - Filter Syntax

- How to filter specific host :

```
host myhostname
dst host myhostname
src host myhostname
```

- How to filter specific ports :

```
port 111
dst port 111
src port 111
```

BPF - Filter Syntax

- How to filter specific net :

```
net 192.168
dst net 192.168
src host 192.168
```

- How to filter protocols :

```
ip proto \tcp
ether proto \ip
```

BPF - Filter Syntax

- Combining expression :

```
&& -> concatenation
not -> negation
|| -> alternation (or)
```

- Offset notation :

```
ip[8] Go the byte location 8 when not specified
      check 1 byte
tcp[2:2] Go the byte location 2 and read 2 bytes
tcp[2:2] = 25 (similar to dst port 25)
Matching (detailed after) is also working tcp[30:4] = 0xDEAD
```

BPF - Filter Syntax

- Offset notation and matching notation (what's the diff?):

```
ip[22:2]=80
tcp[2:2]=80
ip[22:2]=0x80
tcp[2:2]=0x80
```

BPF - Filter Syntax

- Using masks to access "bits" expressed information like TCP flags:

```
+-+-+-+-+-+-+-+-+
|C|E|U|A|P|R|S|F|
|W|C|R|C|S|S|Y|I|
|R|E|G|K|H|T|N|N|
+-+-+-+-+-+-+-+-+
```

```
tcp[13] &9 = 1
tcp[13] &1 = 1
tcp[13] &41 = 41
```

BPF - Filter Syntax

- If you don't want to match every bits, you have some variations.
- Matching only some bits that are set :

  `tcp[12] &9 != 0`
- If you want to match the exact value without the mask :

  `tcp[12] = 1`

BPF - Filter Syntax

- Using masks to access "bits" expressed information like IP version:

```
+-+-+-+-+-+-+-+-+
|Version|  IHL  |
+-+-+-+-+-+-+-+-+
```

```
ip[0] & 0xf0 = 64
ip[0] & 0xf0 = 96
```

BPF - Filter Syntax on Payload

- Matching content with a bpf filter. bpf matching is only possible on 1,2 or 4 bytes. If you want to match larger segment, you'll need to combine filter with &&.
- An example, you want to match "GE" string in a TCP payload :

```
echo -n "GE" | hexdump -C
00000000  47 45                    |GE|
sudo tcpdump -s0 -n -i ath0 "tcp[20:2] = 0x4745"
```

Libpcap - a very quick introduction

- How to open the link-layer device to get packet :

```
pcap_t *pcap_open_live(char *device, int snaplen,
                       int promisc, int to_ms,
                  char *ebuf)
```

- How to use the BPF filtering :

```
int pcap_compile(pcap_t *p, struct bpf_program *fp,
                 char *str, int optimize,
          bpf_u_int32 netmask)
int pcap_setfilter(pcap_t *p,
                   struct bpf_program *fp)
```

Libpcap - a very quick introduction 2/2

- How to capture some packets :

```
u_char *pcap_next(pcap_t *p, struct pcap_pkthdr *h)
```

- How to read the result (simplified) from the inlined structs :

```
sniff_ethernet addr
sniff_ip addr + SIZE_ETHERNET
sniff_tcp addr + SIZE_ETHERNET
               + {IP header length}
payload addr + SIZE_ETHERNET
             + {IP header length}
             + {TCP header length}
```

Libpcap libraries

You don't like C and want to code fast for the workshop...
Here is a non-exhaustive list of libcap (and related) binding for other languages :

- *Net::Pcap - Perl binding*
- *pcap ruby - Ruby binding with a nice OO interface*
- *pylibpcap - Python binding*
- *MLpcap - ocaml binding ;-)*
- *...*

Libpcap tools

- tcpdump, tcpslice (mergecap)
- ngrep (you can pass regex search instead of offset search)
- tcpflow
- tcptrace
- ipsumdump
- tcpflow

Digging in real pcaps

The common capture that will be used in this workshop :
MD5 65ca24413de7ab0ad6423ed2b6329056 capture.cap
MD5 992cb16347bb963242a18560892c4df2 pcap_2012-09-16.zip
MD5 db066fcd23e50534978236de5fb8977 vibrowa.cap

- Where to start? Focus on little events? huge events?
- How to cut the capture? Slicing by date? by size?
- You can use any of the tools proposed but ...
- ... you can build your own tools to ease your work.
- Time reference is a critical part in forensic analysis.
- Be imaginative.

Q and A

- Thanks for listening.
- a@foo.be