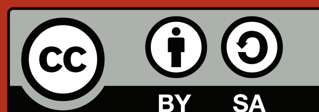
What is Computer Science?

An Information Security Perspective

Daniel Page ⟨dan@phoo.org⟩ and Nigel P. Smart ⟨csnps@bristol.ac.uk⟩

git # b4055dd3 @ 2019-05-13



A TOUR OF COMPUTER NETWORKING AND CRYPTOGRAPHY-BASED NETWORK SECURITY

The term **network security** is a catch-all for a *very* broad field. In essence, it describes techniques and technologies relating to secure use of and communication over a network (e.g., with network-attached resources). On one hand, network security implies various specific challenges, and so requires a specific set of related background knowledge. On the other hand, some challenges within this context are fairly generic; the need for confidentiality and integrity of data, plus authentication of parties could also be evident in non-networked information systems for example.

As a result, one can use network security as a specific vehicle to study these more general concepts. At least two benefits, versus an alternative, stem from doing so:

1. it can offer a practical explanation of concepts we all routinely make use of, plus
2. relate this practical exploration to any theoretical introduction of the same concepts you might have encountered.

This is a (very) short tutorial which explores these points, with specific focus on security-related topics. Said topics, and hence the remit of the tutorial as a whole, are limited in two important ways. First, the content tries to follow a similar ethos as elsewhere in the book by using BASH commands to explain each topic, e.g., those from Chapter 7 and Chapter 10. More so than elsewhere however, these examples are dependent on how the Operating System (OS) works. As a result, and though the concepts themselves are of course more general and can be translated, the examples are specific to how networking is dealt with by the Linux kernel [15]. Second, we limit examples to commands you can issue as a normal user: a huge range of interesting topics (e.g., packet filtering and analysis [10, 22]) are possible if you can act as a privileged user (i.e., `root` or similar [30]), but are outside the remit of this more basic introduction. To mitigate both limitations, note that plenty of resources provide further information with at least one free online book

http://en.wikibooks.org/wiki/Linux_Networking

for example.

To get the most out of what follows, the recommended approach is to first read through Section 1 which introduces important background concepts and terminology, *then* each subsequent Section (more or less in any order you want).

1 Some basic, background concepts and terminology

Probably without realising it, you already have at least an intuitive grip of some standard network concepts: after all, you routinely *use* them even if unknowingly. Since we need somewhere to start, the following is a brief recap of such concepts and terminology then used throughout:

- A **network stack** allows processes executing on each **host** (or computer, sometimes termed a **node**) to access the network. The internal organisation of a concrete network stack often follows the **OSI model** [21], or similar, in the sense that a series of layers each provide different types of functionality to a process.

  The lower-layers often represent physical hardware devices that form **network interfaces** (i.e., physical connections to the network infrastructure) via **Network Interface Controllers (NICs)** [20]. The upper-layers are more typically realised in software as part of the kernel (due, for example, to the need for direct and protected access to the NIC). These layers will manage protocols such as the **Internet Protocol (IP)** [14] and **Transmission Control Protocol (TCP)** [32].

  Rather than view it in terms of different layers, we take a more abstract and simplistic approach: we ignore which role the IP or TCP layer has for example, simply assuming a single monolithic network stack within the kernel does *everything*.

- An IP-based network, such as the Internet, is packet switched. To send a variable-length message M from some source host to a target (or destination), it is first split into one or more fixed-length **packets**. This can be roughly formalised by saying

$$M \mapsto P = \langle P_0, P_1, \ldots P_{n-1} \rangle,$$

  meaning a message M maps to a sequence P of n packets. Each of these packets is **routed** independently through the network via a series of intermediate **hops** (rather than in a single, direct hop from source to target), then reassembled back into M once they all reach the target.

- An **IP address** is a unique numerical identifier given to each host on an IP-based network. Depending on the type of network (e.g., IPv4 or IPv6) the IP address format might differ, but here we assume it is a 4-tuple of 8-bit bytes. More formally for example, the IP address

  `137.222.102.8`

  could be represented by the tuple

$$A = (A_0, A_1, A_2, A_3) = (8, 102, 222, 137)$$

  where each $A_i \in \{0, 1, \ldots, 255\}$. Some IP addresses are reserved for special purposes, one example being `127.0.0.1` which identifies the **local host**, i.e., the computer you are issuing commands on.

- Imagine we have an IP address written as

$$A = (A_0, A_1, A_2, A_3).$$

  If we fix A_1, A_2 and A_3, this specifies a so-called class C **sub-net** [29] (or "smaller network"): each of the 256 possible options for A_0 can be used to identify a host on said sub-net. Sometimes, a short-hand such as

  `137.222.102.0/24`

  is used in this context. Here, the number after the slash tells us how much of the IP address is fixed: 24 bits of the address are fixed (matching A_1, A_2 and A_3), and 8 bits can be varied (matching A_0).

- A related idea is the application of an **address mask** to some IP address. Imagine we have a mask

$$B = (0, 255, 255, 255)$$

  for example. The idea is to combine M and A by using the AND function (outlined in Chapter 2) to get

$$A' = A \wedge B = (0, A_1, A_2, A_3)$$

  because zero AND *any* A_0 will produce zero as a result in A'_0. Thus, we can use M to ignore some parts of A: if we start with

  `137.222.102.8`

  and apply our mask, we get

  `137.222.102.0`

  as a result.

- The **Domain Name System (DNS)** [8] is an infrastructure that supports associations between IP addresses and more easily human-readable string-based identifiers (or names). Translation of a **DNS name** into an IP address is termed **DNS resolution**; for example, we might resolve the DNS name `www.cs.bris.ac.uk` into the IP address `137.222.102.8`.
- Imagine a process (say **A**) on one host needs to communicate a message to a process (say **C**) on another host. Clearly the host **C** is executing on is identified by an IP address; as such, we can be confident packets sent by **A** will at be routed to the correct target. But how does that target know to deliver the packets to **C** rather, for example, than other processes (say **B** or **D**)?

  This problem is solved by the network stack (more formally as part of the TCP protocol) maintaining a set of **ports** [25], each of which has a numerical identifier between 1 and 65535. A given process can send and receive data to and from a specific port via the network stack. As long as **A** marks packets it sends with the port number **C** is using, the packets always end up in the right place.

  Standard **services** are often (pre-)assigned standard ports (e.g., HTTP normally uses port 80, SMTP uses port 25 and so on). You might find cases were a port number is appended to an IP address: when we write

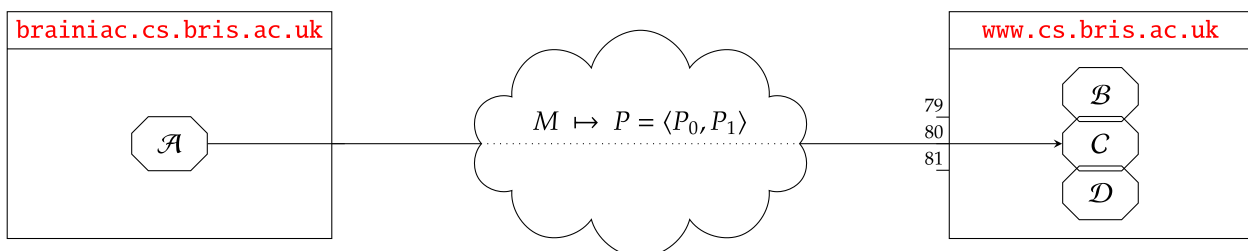
  `137.222.102.8:80`

  or

  `www.cs.bris.ac.uk:80`

  the number after the colon refers to port 80 (specified using the IP address `137.222.102.8` or DNS name `www.cs.bris.ac.uk`).

Consider a somewhat simplified example, described using a diagram: we use a similar style to illustrate and explain scenarios throughout the tutorial. The idea here is to capture some (hopefully most) of the concepts above more coherently:



The diagram illustrates two processes, **A** and **C**, that represent a web-browser and web-server respectively; **A** executes on a host whose DNS name is `brainiac.cs.bris.ac.uk`, **C** on a host whose DNS name is `www.cs.bris.ac.uk`. Imagine **A** wants to send a 2kB HTTP request M to **C**. First, **A** resolves the DNS name `www.cs.bris.ac.uk` to the IP address `137.222.102.8`; it sends M via the network stack on `brainiac.cs.bris.ac.uk`. The network stack then splits M into a 2-packet sequence $P = \langle P_0, P_1 \rangle$, before sending each of the 1500B packets via an Ethernet [9] card into the network. In this case, P_1 is not likely to be full: although two packets are required to communicate the whole of M, they could cope with messages up to $2 \cdot 1500\text{B} = 3000\text{B}$ in length. Either way, intermediate hosts then forward P_0 and P_1 until they reach the target host, where the network stack on `www.cs.bris.ac.uk` receives them: it reassembles the packets into M, and presents them to the process using port 80, which is **C**, for it to use somehow.

The challenge now is to explore these concepts in a practical and concrete way. There are *lots* of related commands we could look at, but four or five alone allow a fairly good coverage and represent the focus in what follows.

1.1 Exploring IP and DNS information for a host

1.1.1 Using `hostname`

The `hostname` command provides access to a limited amount of information about the local host, mainly related to IP and DNS addresses. It does this by invoking system calls such as `gethostname` on our behalf, each of which essentially asks the kernel a question about how the network stack is configured. Consider the following example

```
bash$ hostname
brainiac
bash$ hostname -f
brainiac
bash$ hostname -s
```

```
brainiac
bash$ hostname -d
bash$ hostname -i
127.0.1.1
bash$
```

which was executed on the host brainiac.cs.bris.ac.uk. Notice that

- the -f option prompts hostname to print the full-qualified DNS name of the local host, which matches the default,
- the -s option prompts hostname to print the host name of the local host (i.e., the first part of the fully-qualified DNS name),
- the -d option prompts hostname to print the domain name of the local host (i.e., the last part of the fully-qualified DNS name), and
- the -i option prompts hostname to print the IP address of the local host.

The last command produces what might seem a surprising result: *two* IP addresses are printed, suggesting that both 137.222.102.127 *and* 127.0.0.1 be used to identify this host. Several sensible reasons explain why this is reasonable in general; for example, maybe the host has two network interfaces (e.g., two network cards, perhaps one wired and one wireless). In this specific case however, the IP address 127.0.0.1 *always* refers to the local host: externally this host is identified by 137.222.102.127, but internally it can be referred to as either 137.222.102.127 (which might arguable change, based on the network configuration) *or* 127.0.0.1 (which is always fixed).

1.1.2 Using host and dig

hostname is all well and good, but has a drawback in that we can only use it to find out information about the *local* host! This is not a limit however: DNS is a distributed, networked system so we can query it for information *about* any host *from* any host. Among several alternatives, two in particular allow some useful examples:

- host is simple to use but focuses on a the basic remit of translating DNS names to IP addresses and vice versa.
- dig is more complex (it has more options for example) but also more powerful: DNS servers house DNS records for each host that include more than simply a mapping between name and IP address, and dig enables a more complete exploration of this meta-data.

Consider the following examples, which both retrieve DNS-related information about the host brainiac.cs.bris.ac.uk:

```
;; ANSWER SECTION:
brainiac.cs.bris.ac.uk. 86400   IN      CNAME   it057374.users.bris.ac.uk.
it057374.users.bris.ac.uk. 86400 IN     A       137.222.102.127

;; AUTHORITY SECTION:
bris.ac.uk.             86400   IN      NS      irix.bris.ac.uk.
bris.ac.uk.             86400   IN      NS      ncs.bris.ac.uk.
bris.ac.uk.             86400   IN      NS      ns3.ja.net.

;; ADDITIONAL SECTION:
irix.bris.ac.uk.        86400   IN      A       137.222.8.143
irix.bris.ac.uk.        86400   IN      AAAA    2001:630:e4:82:137:222:8:143
ncs.bris.ac.uk.         86400   IN      A       137.222.8.142
ncs.bris.ac.uk.         86400   IN      AAAA    2001:630:e4:82:137:222:8:142
ns3.ja.net.             4533    IN      A       193.63.106.103
ns3.ja.net.             4533    IN      AAAA    2001:630:0:46::67

;; Query time: 1 msec
;; SERVER: 127.0.1.1#53(127.0.1.1)
;; WHEN: Thu Jun 22 08:57:53 BST 2017
;; MSG SIZE  rcvd: 289

bash$
```

The difference is immediately obvious: the output from host is simply the IP address of brainiac.cs.bris.ac.uk, but dig provides much more information (which may or may not be useful). What about the other direction, i.e., looking-up a DNS name given an IP address? Using the -x option, dig can also perform this type of *reverse* DNS look-up:

```
;127.102.222.137.in-addr.arpa.  IN      PTR

;; ANSWER SECTION:
127.102.222.137.in-addr.arpa. 86400 IN  PTR     it057374.users.bris.ac.uk.

;; AUTHORITY SECTION:
222.137.in-addr.arpa.   86400   IN      NS      ns3.ja.net.
222.137.in-addr.arpa.   86400   IN      NS      irix.bris.ac.uk.
222.137.in-addr.arpa.   86400   IN      NS      ncs.bris.ac.uk.

;; ADDITIONAL SECTION:
irix.bris.ac.uk.        86400   IN      A       137.222.8.143
irix.bris.ac.uk.        86400   IN      AAAA    2001:630:e4:82:137:222:8:143
ncs.bris.ac.uk.         86400   IN      A       137.222.8.142
ncs.bris.ac.uk.         86400   IN      AAAA    2001:630:e4:82:137:222:8:142
ns3.ja.net.             4423    IN      A       193.63.106.103
ns3.ja.net.             4423    IN      AAAA    2001:630:0:46::67

;; Query time: 1 msec
;; SERVER: 127.0.1.1#53(127.0.1.1)
;; WHEN: Thu Jun 22 08:59:43 BST 2017
;; MSG SIZE  rcvd: 289

bash$
```

1.2 Checking a host is active using ping

We use the term "ping" as part of everyday life: to ping someone means to check on them, or remind it that something needs to be done, or just get their attention. The `ping` command applies a similar idea to a target host, testing whether it is operational (i.e., whether it can be connected to, meaning it is reachable via the network). It does this by sending special messages to the target host, formally these are **Internet Control Message Protocol (ICMP)** [13] echo requests. A corresponding reply (or absence thereof) will allow `ping` to and interpret as meaning the target is operational (or not).

Consider the following three examples, wherein `ping` is limited to sending five echo request (by default it will continue until terminated) using the `-c` option:

```
bash$ ping -c 5 foo
ping: unknown host foo
bash$ ping -c 5 toybox.cs.bris.ac.uk
ping: unknown host toybox.cs.bris.ac.uk
bash$ ping -c 5  snowy.cs.bris.ac.uk
PING snowy.cs.bris.ac.uk (137.222.103.3) 56(84) bytes of data.

--- snowy.cs.bris.ac.uk ping statistics ---
5 packets transmitted, 0 received, 100% packet loss, time 4031ms

bash$
```

In the first case the host `foo` does not exist, so `ping` cannot translate the DNS name into a usable target IP address for the ICMP packets. The second case is more interesting: `toybox.cs.bris.ac.uk` is translated into the IP address `137.222.102.74`, but ICMP packets sent cannot be routed to the target for some reason. In the third case, `ping` demonstrates that `snowy.cs.bris.ac.uk` (whose IP address is `137.222.103.3`) is operational: among other information, notice that

- for each message sent by `ping`, the round-trip time (i.e., the time taken to send the message and receive a response) is listed, and
- once `ping` as finished, some statistics are produced that in some sense describe the connection quality: the number of messages which failed to produce a response is recorded for example, as are maximum, minimum and average round-trip times.

1.3 Exploring the path between hosts using traceroute

We already mentioned that messages, in the form of packets, are routed to from source to their target by making hops between intermediate hosts. A earlier network, the **Advanced Research Projects Agency Network (ARPANET)** [2], pioneered this approach. Among the original design objectives, scalability and reliability (versus a network with direct host-to-host connections) were central: the former is improved by removing for the need for n^2 potential direct host-to-host connections between n hosts, the latter is improved by virtue of the potential to change or update the routing strategy (e.g., to avoid some host that has failed).

The `traceroute` command can be used to give information about the route (i.e., the hops, or path taken between hosts) from the local host (acting as the source) to a target host. Like `ping` it uses ICMP echo requests, but also the **Time-To-Live (TTL)** feature. The idea is that if the TTL for some packet is set to t, if the target host is not reached after t hops then an error is sent back to the source; `traceroute` harnesses this feature by

An aside: security issues with the ICMP echo (AKA ping) request and reply process.

The word "usually" in the description of how a target host *should* respond to an ICMP echo request might sound a little vague: the point is that the host can opt out by not producing an ICMP echo reply, meaning `ping` will assume it is non-operational. Why might it do this? One reason is that the host wants to avoid being (easily) discovered, but others exist as well:

1. It takes a some resource (time, bandwidth etc.) in order to process an ICMP echo request and generate a reply; the level of resource might be small, but is certainly non-zero. With this in mind, one class of **Denial-of-Service (DoS)** attack [6] is the so-called **ping flood** [23]. The idea is simple: swamp some target host with *lots* of ICMP echo requests, meaning it has too little time or bandwidth to operate as normal (i.e., execute legitimate processes).
2. Like any software, the network stack on a target host might contain a bug. In fact, several such bugs relating to ICMP echo requests have been found (in more than one OS) and exploited. One example is the so-called **Ping-of-Death (PoD)** [24] where an oversized ICMP echo request is sent by an attacker to the target host; if the host cannot cope with this gracefully (due to the bug), the resulting error could actually crash the target and force it to reboot!

Both attacks are powerful in the sense they can be mounted remotely over the network: the attack does not need access to log into the target host, for example.

An aside: cryptic hosts and routes.

When using `traceroute`, there are at least two types of entry in the output which could be confusing:

1. When information about a host cannot be determined, it is replaced by asterisk character. Various reasons exist for such an event, but the most basic is that the host failed to reply within the time limit; this can be extended using the `-w` option. Another could be a failure to complete a reverse DNS look-up (i.e., the resolution of a DNS name from an IP address): this process can be avoided using the `-n` option.
2. Sometimes, hosts appear which you might not expect. In the example, we get something *other* than `www.bbc.co.uk`: why is this? It might be a result of **load balancing** [17]: if a single host could not cope with the volume of accesses to a web-site for example, a *pool* of hosts would be tasked with managing the workload between them.

successively increasing the TTL (sending a so-called probe), so at each step it discovers the next hop made from source to target.

Consider an example where the local host is brainiac.cs.bris.ac.uk, and the target is www.bbc.co.uk:

```
 8  * * *
 9  * * *
10  * * *
11  * * *
12  * * *
13  * * *
14  * * *
15  * * *
16  * * *
17  * * *
18  * * *
19  * * *
20  * * *
21  * * *
22  * * *
23  * * *
24  * * *
25  * * *
26  * * *
27  * * *
28  * * *
29  * * *
30  * * *
bash$
```

Of course `traceroute` accepts numerous other options that each allow more advanced behaviour, but even here we get

- a list of hosts that the packages are routed via, implying the number of hops required, and
- the latency of each hop (one entry per-host, per-probe of that host of which there could be several) between hosts, meaning the delay associated with communication at that point.

In this example, notice that packets start within a network associated with the University of Bristol, then travel over the so-called **Joint Academic NETwork (JANET)** which rough acts as an **Internet Service Provider (ISP)** for Universities in the UK, before reaching a network associated with the BBC.

1.4 Inspecting network configuration using `netstat`

On most Linux distributions, the file `/etc/services` holds a list of standard service names and their corresponding port numbers. It does not tell us anything about which services are actually active however. The `netstat` command can be used to do this, and indeed act as a general way to explore the active network configuration (on the local host). In particular, it provides information and statistics about

1. network interfaces (i.e., connections from the local host to a given network provided via wired or wireless network cards) via the `--interface` option,
2. network routing tables (i.e., where the local host will send packets via in order to reach a given target host) via the `--route` option,
3. network connections (e.g., incoming and outgoing, active and inactive connections) via options such as `--listening`, and
4. network protocols (e.g., how much data has been sent specifically via TCP connections) via options such as `--tcp`, `--udp` and `--raw`.

It should be noted that a lot of the same information will be exported by the kernel via the `/proc/net/` file system; for example, `/proc/net/route` lists the routing tables. As such, you can think of `netstat` as simple a standard way to collect this and present it in a more human-readable form. Also note that throughout, the option `--numeric` is used to force output in a numeric form (e.g., as an IP address rather than DNS name).

1.4.1 Network interfaces and routing tables

Consider examples of the first two use-cases:

```
bash$ netstat --numeric --interface
Kernel Interface table
Iface    MTU Met    RX-OK RX-ERR RX-DRP RX-OVR    TX-OK TX-ERR TX-DRP TX-OVR Flg
eth0       1500 0     302227      0      0 0       137090      0      0      0 B
MRU
lo        65536 0      95342      0      0 0        95342      0      0      0 L
```

```
RU
wlp4s0     1500 0    151111      0      0 0       115148      0      0      0 B
MRU
bash$ netstat --numeric --route
Kernel IP routing table
Destination     Gateway         Genmask         Flags   MSS Window  irtt Iface
0.0.0.0         10.70.2.250     0.0.0.0         UG        0 0          0 eth0
0.0.0.0         172.23.255.254  0.0.0.0         UG        0 0          0 wlp4s0
10.70.2.0       0.0.0.0         255.255.255.0   U         0 0          0 eth0
137.222.7.170   172.23.255.254  255.255.255.255 UGH       0 0          0 wlp4s0
137.222.7.214   10.70.2.250     255.255.255.255 UGH       0 0          0 eth0
169.254.0.0     0.0.0.0         255.255.0.0     U         0 0          0 eth0
172.23.0.0      0.0.0.0         255.255.0.0     U         0 0          0 wlp4s0
bash$
```

In the former example, two entries are listed for `eth0` (a wired Ethernet [9] interface) and `lo` (an artificial form of loop-back [18] interface, existing within the local host only rather then acting as a connection to a physical network). For each entry, we see information including

- the **Maximum Transmission Unit (MTU)**, which is basically the maximum size of packets (in bytes) one can send over the interface,
- packet statistics such as RX-OK and RX-ERR for example, listing the total number of correctly and incorrectly formed packets received, and
- a set of 1-character flags detailing the type and status of the interface (e.g., the 'L' character means loop-back, and 'U' means the interface is active).

In the latter example, the routing table is displayed: each entry forms a rule used by the kernel to decide where packets should be sent (formally this dynamic approach can be supplemented by a static routing policies, but we ignore this). When the kernel is tasked with routing a packet to some target IP address, say

$$T = (T_0, T_1, T_2, T_3),$$

it steps through each rule in turn:

1. First it applies the mask; taking the first rule as an example, the cited mask
$$M = (0, 255, 255, 255)$$
means we end up with
$$T' = M \wedge T = (0, T_1, T_2, T_3).$$
2. Then, it compares T' with the destination field: if they match, the packet is forwarded to the gateway address if need be.

So, our routing table implies the following:

- A packet for the target `137.222.102.149` matches the first rule (since `137.222.102.149` AND'ed with the mask `255.255.255.0` is `137.222.102.0`) for example, but the gateway entry `0.0.0.0` means it does not need to be forwarded,
- A packet for the target `169.254.1.1` matches the second rule (since `169.254.1.1` AND'ed with the mask `255.255.0.0` is `169.254.0.0`) for example, but the gateway entry `0.0.0.0` means it does not need to be forwarded: this is a special link-local [16] range.
- All other packets match the third rule (since the destination `0.0.0.0` is a default meaning any packet), and are forwarded to `137.222.102.250` via the `eth0` interface (then presumably onward from there).

1.4.2 Network connections

The third use-case for `netstat` is as a means of checking which ports are currently in use, and what for. There are numerous reasons to do this: for example, perhaps you want to disable any services you do not need (as a form of optimisation for both performance and security), or perhaps you want to execute a service but first need to check whether the port is already in use. Either way, `netstat` can identify ports which are listening for connections:

```
bash$ netstat --numeric --tcp --listening
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 127.0.1.1:53            0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:22              0.0.0.0:*               LISTEN
tcp6       0      0 :::22                   :::*                    LISTEN
bash$ cat /etc/services | grep  '111/tcp'
sunrpc          111/tcp         portmapper      # RPC 4.0 portmapper
kx              2111/tcp                        # X over Kerberos
bash$ cat /etc/services | grep '1012/tcp'
bash$
```

Each entry details a (potential) connection between the local host and some remote host; the information includes

- the protocol used (in this case, the `--tcp` option limits this to connections using TCP only),
- a count of the number of bytes in receive and send queues (which essentially represent data which has not yet been processed by the local or remote host),
- the local address (meaning the address on the local host, including the port number),
- the remote (or foreign) address (meaning the address of the remote host),
- the connection state (in this case, the `--listening` option limits this to connections in the listening state only).

Various obvious cases stand out in our example (the local host is clearly listening for SSH and SMTP connections on ports 22 and 25), but it also includes some less obvious cases; we can try to identify these using `/etc/services`, but this is not a definitive list. Port 111 seems to relate to the **Remote Procedure Call (RPC)** [26] system for instance, but port 1012 is not included (this could potentially be in use by a non-standard user program for example).

1.4.3 Network protocols

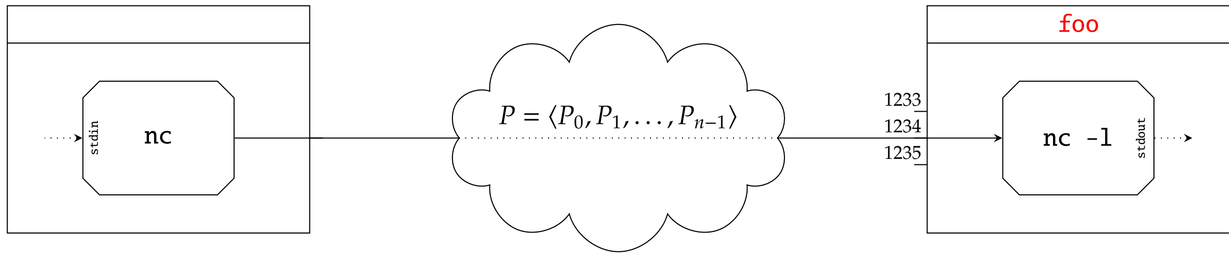
The forth and final use-case for `netstat` is to dump statistics relating to a particular network protocol. For example, the following

```
    294 ICMP messages sent
    0 ICMP messages failed
    ICMP output histogram:
        destination unreachable: 289
        echo request: 5
IcmpMsg:
        InType3: 289
        InType11: 6
        OutType3: 289
        OutType8: 5
UdpLite:
IpExt:
    InMcastPkts: 263
    OutMcastPkts: 4490
    InBcastPkts: 42
    OutBcastPkts: 42
    InOctets: 353173553
    OutOctets: 35216838
    InMcastOctets: 23980
    OutMcastOctets: 870899
    InBcastOctets: 2132
    OutBcastOctets: 2132
    InNoECTPkts: 411726
bash$
```

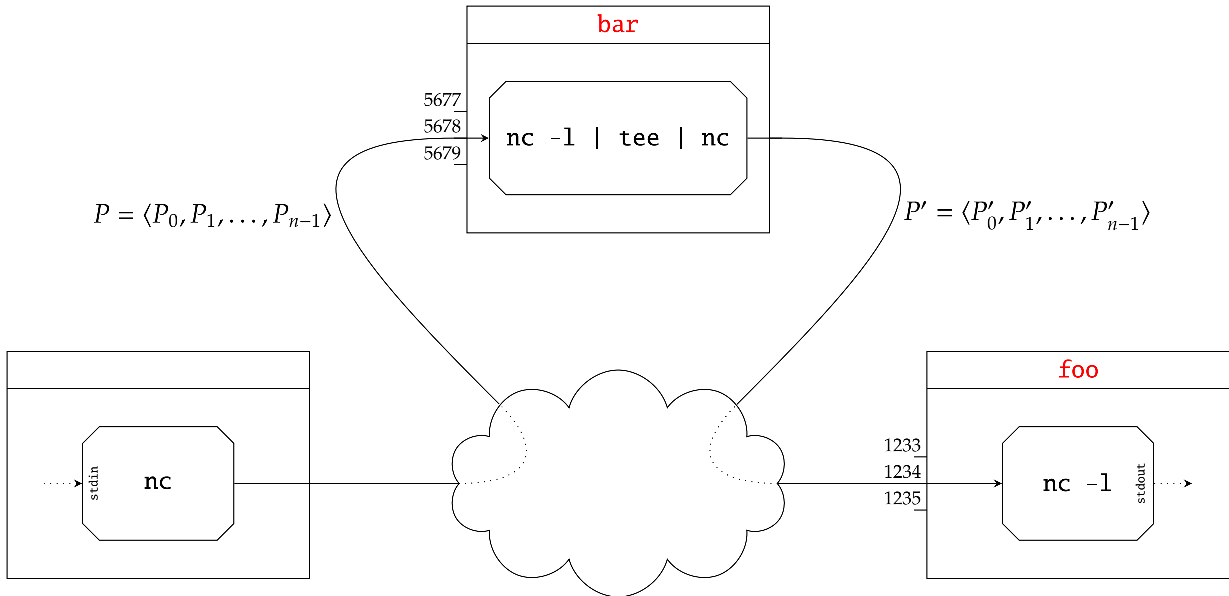
lists raw packet information, i.e., at the IP rather than TCP or UDP level, by virtue of the `--raw` option.

2 Networked inter-process communication using netcat

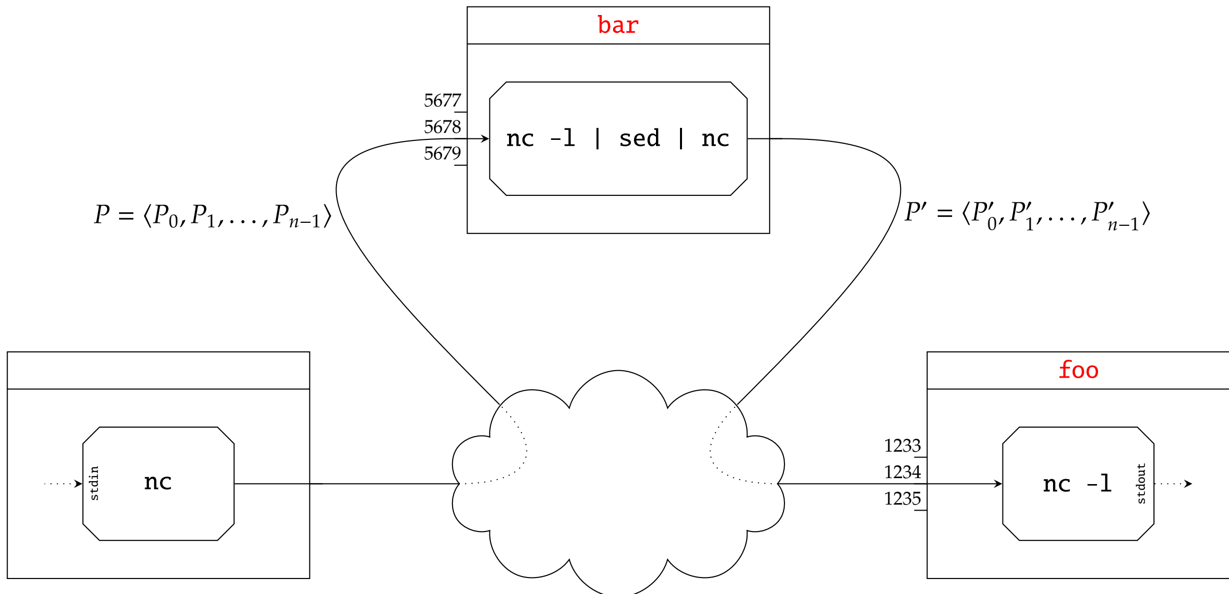
netcat (represented by the command `nc`, and literally a networked version of `cat`) is often described as a “swiss-army knife” of networking, since it can act as a solution for (or within) such a wide range of tasks. In the following, we use `nc` within the context of two such tasks in order to illustrate host-to-host communication; this moves the discussion up a level, with less emphasis on the underlying network stack and more on use at an application level.



(a) *Sending packets directly (i.e., via the network only).*



(b) *Sending packets via a passive man-in-the-middle.*



(c) *Sending packets via an active man-in-the-middle.*

Figure 1: *Three scenarios describing use of* `nc` *to send from a source host to a target host.*

2.1 A simple messaging system

Arguably the simplest use of `nc`, and certainly a good way to start, is to connect the `stdin` or `stdout` streams attached to a given process with a network port instead. More concretely, imagine a task that involves the combination of two processes: we might normally combine the processes *locally* using a pipe (e.g., via a command pipeline). By using `nc` this combination can be *remote*, meaning the pipe is essentially realised by the network. So assume we want to connect `nc` to port 1234 on some host whose DNS name is `foo`, replicating Figure 1a. We can operate the command in two different modes:

1. On `foo` itself, we can use `nc` as a server (meaning it waits or listens for a connection) via

   ```
   nc -l foo 1234
   ```

   noting the `-l` option which specifies this mode. Used as such, `nc` will read input from the port and write it as output to `stdout`.

2. On any host (perhaps `foo` as well, or some other host), we can use `nc` as a client (meaning it initiates a connection) via

   ```
   nc foo 1234
   ```

   In contrast to the above, `nc` will now read input from `stdin` and write it as output to the port.

Diagrammatically, execution of the two commands can be viewed as implying the scenario in Figure 1a. Illustrating these commands being executed from the command-line is a little more tricky that some others, due to the level of (concurrent) user interaction. So instead of relying on an example, step through the following tasks to see how this works yourself:

Implement (task #1)

Open two terminal windows, referred to as terminal #1 and #2. Strictly in order, *first*

1. in terminal #1, execute the command

   ```
   nc -l localhost 1234
   ```

   which acts as the server, *then*

2. in terminal #2, execute the command

   ```
   nc localhost 1234
   ```

   which acts as the client

so both commands refer to the local host via the DNS name `localhost`. Once connected, this produces a messaging system (akin to the UNIX `talk` [31] command) between the terminals: whatever you type into terminal #2 will be displayed on terminal #1. You can terminate the connection either via Ctrl-C (forcibly terminating the process) or Ctrl-D (marking the end of input, at which point the process will terminate naturally) on either terminal.

Implement (task #2)

We *could* replace `localhost` with *any* IP address or DNS name, allowing extension from local-only to remote communication. Either

1. log terminal #2 into a second host (using SSH for example), or

2. find a friend already using a second host (e.g., the person next to you in the lab) and collaborate with them.

Remember that you can find the DNS name and IP address of either host using commands outlined in Section 1.1.

Repeat Task 1 but for terminal #2 acting as the client, replace `localhost` with a reference to the host you run the server on (i.e., either the IP address or DNS name); note the communication is now genuinely via the network (since the client and server are different computers).

```
>
>
> ' | nc www.cs.bris.ac.uk 80 | head -n 50
HTTP/1.1 400 Bad Request
Date: Thu, 22 Jun 2017 08:12:40 GMT
Server: Apache/2.2.22 (Debian)
Vary: Accept-Encoding
Content-Length: 313
Connection: close
Content-Type: text/html; charset=iso-8859-1
Set-Cookie: BIGipServer~SysOps~csweb.app~csweb=3573551788.0.0000; path=/

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>400 Bad Request</title>
</head><body>
<h1>Bad Request</h1>
<p>Your browser sent a request that this server could not understand.<br />
</p>
<hr>
<address>Apache/2.2.22 (Debian) Server at w-tc-p6.cs.bris.ac.uk Port 80</address
>
</body></html>
bash$
```

Figure 2: *Example output from* `nc` *when used as a simple web-browser to load the web-page* `index.html` *from* `www.cs.bris.ac.uk`*.*

2.2 Some simple HTTP-based applications

At a (very) basic level, a web-browser simply reads and writes data to and from a port on some web-server (using the HTTP protocol [12]) and displays the result. On the other side, a web-server is just a process running on some host that receives input (i.e., requests, or commands again using the HTTP protocol) and produces output (i.e., responses, or content). The fact we have already used `nc` to receive and produce input and output over the network suggests it can be used as a (very) basic web-browser and/or web-server: this is exactly our goal in the following.

2.2.1 A web-browser

Consider the following command pipeline:

```
echo -n -e 'GET /index.html HTTP/1.0\r\n\r\n' | nc www.cs.bris.ac.uk 80
```

The right-hand half should be familiar in that we ask `nc` to connect to port 80 of `www.cs.bris.ac.uk`. Rather than read input from the terminal (i.e., have someone type it) however, `nc` reads input from `echo` instead: all output the invocation of `echo` writes to `stdout` is fed, via the pipe, as input on `stdout` to the invocation of `nc`.

The `-n` and `-e` options prompt `echo` to avoid producing a trailing new line automatically, but expand the escaped characters we specify into carriage return and new line respectively. As a result, `echo` produces

```
GET /index.html HTTP/1.0
```

followed by a blank line: this is a request for the web-server to supply the file `/index.html` (i.e., `www.cs.bris.ac.uk:80/index.html`). The response produced is illustrated in Figure 2, where use of `head` produces the first 50 lines only (of what is a large file, preceded by the HTTP header).

Implement (task #3)

Reproduce the steps above to download content from

`http://www.google.com/index.html`

using `nc`, then do the same thing with a real web-browser. Depending on where in the world you do this from (e.g., the UK versus the US), the result might be surprising: using output from the former, try to explain (even informally)

- what each line of the HTTP header and content means, and
- what the real web-browser is therefore doing automatically on your behalf.

Research (task #4)

The details of HTTP are normally hidden from you by a web-browser, but in the above we use it *directly* to perform what is termed a GET request (i.e., to get some content). Do some research into other request types (examples include the HEAD, OPTIONS or TRACE request types), then alter the example above to perform at least one different request via port 80 of www.cs.bris.ac.uk.

Research (task #5)

Although you may be more used to GUI-based web-browsers, e.g. Firefox or Chrome, text-based web-browsers also exist: like `nc`, these can be extremely useful when automating tasks from the command-line. Standard examples include the venerable `links` web-browser, and the `wget` content retrieval command. Focusing on the latter, do some research into how `wget` works then use it, in a similar way as above, to again retrieve content from

http://www.google.com/index.html

2.2.2 A (one-shot) web-server

What about the other side of this scenario? Imagine we create a file that represents a web-page we want to serve (via the network) to a web-browser. Given such a file, say `A.txt`, an `nc`-based web-server can be executed as follows

```
{ echo -n -e 'HTTP/1.0 200 OK\r\n\r\n' ; cat A.txt ; } | nc -l localhost 1234
```

where within the command pipeline

1. the left-hand side uses `echo` and `cat` to print the HTTP header and file content respectively; their combined output (which you can think of as merged together as a result of the curly braces around both commands) is piped into

2. the right-hand side, which uses an invocation of `nc` as before: it listens for a connection to port 1234 of the local host, and when one is made it produces the content as required.

Try it out yourself:

Implement (task #6)

Open two terminal windows, referred to as terminal #1 and #2. Strictly in order, *first*

1. create a file called `A.txt`, containing whatever content you want, but perhaps HTML for example,
2. to make things easier, in terminal #1 first create an alias

   ```
   alias S="{ echo -n -e 'HTTP/1.0 200 OK\r\n\r\n' ; cat A.txt ; }"
   ```

   then execute the command

   ```
   S | nc -l localhost 1234
   ```

   which acts as the web-server, *then*
3. to make things easier, in terminal #2 first create an alias

   ```
   alias C="echo -n -e 'GET /index.html HTTP/1.0\r\n\r\n'"
   ```

   then execute the command we used previously

   ```
   C | nc localhost 1234
   ```

   which acts as the web-browser (or client).

This `nc`-based web-server is *very* basic of course. For example once it has satisfied the request by sending `A.txt` to the web-browser it simply terminates; in addition, it makes no difference what request is sent in that the web-browser gets `A.txt` even if it asked for something else.

Implement (task #7)

We can use a BASH loop to resolve the first problem. Assuming reuse of the alias representing the server, repeat Task 6 but use

```
while true ; do S | nc -l localhost 1234 ; done
```

in terminal #1: now the web-server executes forever (at least until terminated using Ctrl-C), meaning it will now deal with multiple requests.

Implement (task #8)

Rather than use `nc` as a web-browser, try using a *real* web-browser such as Chrome or `links` to access the URL

http://localhost:1234/A.txt

connected to the web-server from Task 7. In the web-browser, you should see the content of `A.txt`; in the web-server terminal window you should see the actual HTTP requests sent by the web-browser when trying to access `A.txt`.

3 Using OpenSSL-based cryptographic primitives

You may recognise OpenSSL as an open-source implementation of the **Secure Sockets Layer (SSL)** and **Transport Layer Security (TLS)** protocols [33]. Although you probably use OpenSSL (even if unknowingly) as a component within web-browsers for example, the project provides a very general-purpose library *and* a suite of command-line tools. From a practical perspective, the latter are enormously useful: making use of them to actually *do things* with cryptography can be both instructive and rewarding. As such, the simple aim of this Section is to explore various common tasks one might undertake using OpenSSL from the command-line.

When invoked from the command-line using the `openssl` command, note that the first, compulsory option determines the operation performed: this is following by normal options that further control the operation.

3.1 Symmetric encryption and decryption operations

The most fundamental cryptographic operation is almost certainly encryption (or decryption) of data, with the easiest approach to doing so being use of a block cipher [3]. OpenSSL supports numerous block cipher algorithms, in numerous modes of operation [4]: we can check the AES-based [1] possibilities as follows

```
bash$ openssl list-cipher-commands | grep aes
aes-128-cbc
aes-128-ecb
aes-192-cbc
aes-192-ecb
aes-256-cbc
aes-256-ecb
bash$
```

noting that 128-bit, 192-bit and 256-bit key sizes are possible, as are ECB and CBC modes of operation.

3.1.1 Using files (and streams)

The first task is to retrieve some data (as elsewhere, some Shakespearean text) to experiment with

```
<A.txt 'http://www.gutenberg.org/dirs/etext97/1ws4110.txt'
bash$
```

after which we can encrypt it, then decrypt to get the same result:

```
bash$ openssl enc -e -aes-128-ecb -k 'secret' -in A.txt -out B.txt
bash$ openssl enc -d -aes-128-ecb -k 'secret' -in B.txt -out C.txt
bash$ cat A.txt | od -Ad -tx1 -w16 -N64
0000000
bash$ cat B.txt | od -Ad -tx1 -w16 -N64
0000000 53 61 6c 74 65 64 5f 5f 06 db be 0d 13 bd f0 c9
0000016 bf f3 a7 b3 77 c6 0f 6f d9 fa 0f 0b 7d 14 26 32
0000032
bash$ cat C.txt | od -Ad -tx1 -w16 -N64
0000000
bash$ cmp A.txt C.txt
bash$
```

The first option `enc` dictates use of a block cipher, with the other options providing extra control:

- `-e` (resp. `-d`) specifies that encryption (resp. decryption) of the input should be performed to produce the output,

- `-aes-128-ecb` specifies that the AES block cipher should be used, with a key size of $k = 128$ bits and in ECB mode,
- `-k` specifies the password used for the encryption (resp. decryption) operation, and
- `-in` (resp. `-out`) specifies the input (resp. output) file name.

In common with most uses of, the input (resp. output) file can be replaced by the standard stream `stdin` (resp. `stdout`) by simply removing the option. To get the same result, we might therefore

```
bash$ cat A.txt | openssl enc -e -aes-128-ecb -k 'secret' > B.txt
bash$ cat B.txt | openssl enc -d -aes-128-ecb -k 'secret' > C.txt
bash$ cat A.txt | od -Ad -tx1 -w16 -N64
0000000
bash$ cat B.txt | od -Ad -tx1 -w16 -N64
0000000 53 61 6c 74 65 64 5f 5f 09 79 a6 a6 2e ac 0b ca
0000016 66 62 a4 c8 de 32 ee d4 30 fc 2a c9 4a b0 d2 82
0000032
bash$ cat C.txt | od -Ad -tx1 -w16 -N64
0000000
bash$ cmp A.txt C.txt
bash$
```

Clearly this can, and will be useful later when the command is used in a larger command pipeline. In addition to encryption and decryption, each example includes four extra commands whose purpose is to demonstrate the content of each input and output. Although we know `A.txt` represents ASCII text, it is important to remember that a block cipher will process it as binary data: it reads (resp. writes) 8-bit bytes of data to form the 128-bit plaintext (resp. ciphertext) message block. The first three commands illustrate this by inspecting the first four blocks (64 bytes in total) of each file. Notice the match between `A.txt` and `C.txt`, which is further confirmed by using `cmp` to perform a complete comparison: the lack of output indicates the files are identical, which we expect as decryption should "undo" encryption (under the same key).

In each of the examples above `-k` specifies an ASCII password (i.e., the string "secret"). This might seem odd if you know about block ciphers: specifically, they do *not* actually use a password but rather a k-bit key K. OpenSSL actually forms such a key *from* the password using a **Key Derivation Function (KDF)**. In more detail, the hash function MD5 [19] is used: without a salt [27] value we get

$$K = \text{MD5}(\text{"secret"}) = 5EBE2294ECD0E0F08EAB7690D2A6EE69_{(16)}$$

used as the key, but *with* salt we instead get whatever

$$K = \text{MD5}(\psi \parallel \text{"secret"})$$

produces given a random salt value ψ. In this case (and others) it can be useful to inspect the resulting key, with the `-p` option instructing OpenSSL to do so. For example, in the following (where the output is discarded)

```
<ecb -nosalt -k 'secret' -p -in A.txt -out /dev/null
key=5EBE2294ECD0E0F08EAB7690D2A6EE69
<ecb         -k 'secret' -p -in A.txt -out /dev/null
salt=13DBB4CF803879A2
key=453849F007C0DBE61D059EEE83E79C60
bash$
```

note that we get the same derived key as above from the same password. Alternatively, we could specify the key (and/or IV, where the mode of operation allows one) explicitly: on one hand this is perhaps less user friendly, but on the other hand it allows more direct control over the block cipher. In this example, we set the cipher key to

$$K = 000102030405060708090A0B0C0D0E0F_{(16)}$$

using the `-K` option (plus the `-iv` option to set a zero'ed IV) as follows:

```
<28-cbc -K 000102030405060708090A0B0C0D0E0F -iv 0 -in A.txt -out B.txt
<28-cbc -K 000102030405060708090A0B0C0D0E0F -iv 0 -in B.txt -out C.txt
bash$ cat A.txt | od -Ad -tx1 -w16 -N64
0000000
bash$ cat B.txt | od -Ad -tx1 -w16 -N64
0000000 95 4f 64 f2 e4 e8 6e 9e ee 82 d2 02 16 68 48 99
0000016
bash$ cat C.txt | od -Ad -tx1 -w16 -N64
0000000
bash$ cmp A.txt C.txt
bash$
```

However, if we specify neither a password *or* key then OpenSSL prompts us to first enter then verify a password choice by typing it:

```
bash$ openssl enc -e -aes-128-ecb -in A.txt -out B.txt
enter aes-128-ecb encryption password:
Verifying - enter aes-128-ecb encryption password:
bash$
bash$ openssl enc -d -aes-128-ecb -in B.txt -out C.txt
enter aes-128-ecb decryption password:
bash$
bash$ cat A.txt | od -Ad -tx1 -w16 -N64
0000000
bash$ cat B.txt | od -Ad -tx1 -w16 -N64
0000000 53 61 6c 74 65 64 5f 5f 92 2f 85 63 f1 84 a5 44
0000016 1f ac d3 49 b8 a6 f6 05 d9 60 7f c5 ba a8 74 e6
0000032
bash$ cat C.txt | od -Ad -tx1 -w16 -N64
0000000
bash$ cmp A.txt C.txt
bash$
```

3.1.2 Across the network

The fact OpenSSL can deal with stream-based as well as file-based input and output suggests an interesting next step: we can already use netcat to perform networked communication, so why not do this in a secure manner by also using OpenSSL to encrypt the communicated data? Following the original example in Figure 1a, again consider a host whose DNS name is `foo`. Using `nc` interactively (per the messaging system example) is a little tricky, but simply communicating the contents of a file is *much* easier:

1. On `foo` itself, we can use `nc` as a server (meaning it waits or listens for a connection) via

   ```
   nc -l foo 1234 | openssl enc -d -aes-128-ecb -k 'secret' > B.txt
   ```

   where the output is fed though `openssl` to decrypt it using AES, forming `B.txt`.

2. On any host (perhaps `foo` as well, or some other host), we can use `nc` as a client (meaning it initiates a connection) via

   ```
   cat A.txt | openssl enc -e -aes-128-ecb -k 'secret' | nc foo 1234
   ```

   where the input `A.txt` is fed though `openssl` to encrypt it using AES.

Challenge (task #9)

Replacing `foo` with the local host whose DNS name is `localhost`, as you did before in Task 1, reproduce the steps above to form an encrypted version of the previous messaging system; verify that the file `B.txt` received by terminal #1 matches the file `A.txt` sent by terminal #2.

Clearly the client and server must operate symmetrically wrt. encryption. Imagine, for example, they use

- a different key, or
- a different block cipher (or mode of operation).

First try to explain what *should* happen in theory, and then try the options in practice and see if the result matches what your explanation.

This is in fact *so* easy, it is difficult to see the value in doing so. Put another way, the point of using encrypted communication is to thwart an attack of some sort: without one, why bother?! In Figure 1b we find some motivation, where versus the initial scenario in Figure 1a a man-in-the-middle is now included. In an IP-based network, we have already know packets are routed through intermediate hosts. As a result, it should be no surprise that a host can inspect and/or store any packet routed through it. The following attempts to give a practical illustration of this:

Implement (task #10)

Open three terminal windows, referred to as terminals #1, #2 and #3. Strictly in order, *first*

1. in terminal #1, execute the command

   `nc -l localhost 5678`

   which acts as the server, *then*

2. in terminal #2, execute the command

   `nc -l localhost 1234 | tee P.txt | nc localhost 5678`

   which acts as the passive man-in-the-middle, *then*

3. in terminal #3, execute the command

   `nc localhost 1234`

   which acts as the client.

Once connected, this produces a messaging system as before: a message typed into terminal #3 appears on terminal #1. However, now the passive man-in-the-middle *also* captures messages into a file called `P.txt`.

Implement (task #11)

Reproduce Task 10, but incorporate use of OpenSSL into the command pipelines for terminals #3 and #1 so each message they communicate are encrypted (using AES say). Before this change, the man-in-the-middle could simply read `P.txt`, but now it needs to know (and use) the key to do so.

Verify this: inspect the contents of (the encrypted) `P.txt`, then decrypt it with the key used by terminals #3 and #1.

3.2 Cryptographic hash and MAC operations

In the same way as block ciphers, OpenSSL allows use of numerous hash functions; the following

```
bash$ openssl list-message-digest-commands
md4
md5
rmd160
sha
sha1
bash$
```

again checks the possibilities, which include MD4, MD5 and SHA-1.

3.2.1 Using files (and streams)

First fetching the same data to work with as above,

```
<A.txt 'http://www.gutenberg.org/dirs/etext97/1ws4110.txt'
bash$
```

applying SHA-1 [28], or a Message Authentication Code (MAC) based on it, say HMAC-SHA-1 [11], is fairly straightforward:

```
bash$ openssl dgst -sha1                A.txt
SHA1(A.txt)= da39a3ee5e6b4b0d3255bfef95601890afd80709
bash$ openssl dgst -sha1 -hmac 'secret' A.txt
HMAC-SHA1(A.txt)= 25af6174a0fcecc4d346680a72b7ce644b9a88e8
bash$
```

Of course if we alter `A.txt` somehow, for instance we change all the 'a' characters to 'b', forming `B.txt`, then applying either SHA-1 or HMAC-SHA-1 produces a totally different output:

```
bash$ cat A.txt | tr 'a' 'b' > B.txt
bash$ openssl dgst -sha1                B.txt
SHA1(B.txt)= da39a3ee5e6b4b0d3255bfef95601890afd80709
bash$ openssl dgst -sha1 -hmac 'secret' B.txt
HMAC-SHA1(B.txt)= 25af6174a0fcecc4d346680a72b7ce644b9a88e8
bash$
```

OpenSSL uses a slightly annoying output format in the sense we often only want the hash function digest or MAC tag: this can be extracted using cut, for example

```
bash$ openssl dgst -sha1                A.txt | cut -d ' ' -f 2
da39a3ee5e6b4b0d3255bfef95601890afd80709
bash$ openssl dgst -sha1 -hmac 'secret' A.txt | cut -d ' ' -f 2
25af6174a0fcecc4d346680a72b7ce644b9a88e8
bash$
```

3.2.2 Across the network

Figure 1c offers a third scenario where the previously passive man-in-the-middle (in Figure 1b) now becomes an *active* man-in-the-middle: instead of just taking a copy of packets communicated through it, packets are now *manipulated* somehow before they reach the target.

Implement (task #12)

Revisit Task 10 (with a passive man-in-the-middle based on use of the tee command, without any form of encryption), but alter the command used by terminal #2 from

```
nc -l localhost 1234 | tee P.txt | nc localhost 5678
```

to

```
nc -l localhost 1234 | sed -u -e 's/a/e/g' | nc localhost 5678
```

instead: the sed command (using -u to operate in unbuffered mode) means 'a' characters sent by terminal #3 will now be translated into 'e' before reaching terminal #1. That is, the active man-in-the-middle manipulates the communication rather than simply observing it.

Verify this works by typing some input into terminal #3: note that the output in terminal #1 might differ (where you type an 'a') from that sent by terminal #3.

Implement (task #13)

sed can also process binary input: replace the command

```
nc -l localhost 1234 | sed -u -e 's/a/e/g' | nc localhost 5678
```

in Task 12 with

```
nc -l localhost 1234 | sed -u -e 's/\x00/\x11/g' | nc localhost 5678
```

instead: this means every byte with the value $00_{(16)}$ is now translated into $11_{(16)}$.

Now encrypt communication between terminals #3 and #1 by again using OpenSSL. That is, communicate the contents of a file A.txt from terminal #3 to terminal #1 (via this new man-in-the-middle) and store the content in B.txt, then verify whether A.txt and B.txt match. Manipulation by the active man-in-the-middle should ensure there are some differences (depending on the original file content, and which bytes the man-in-the-middle manipulates).

Without access to `A.txt` *and* `B.txt`, we cannot verify they match (as per Task 13). So how *can* we detect whether or not the communication is manipulated? One way is to employ a MAC. This task is somewhat complicated, but the idea is to start by revisiting Task 1 (without a man-in-the-middle, or encryption), and see how to do this:

Implement (task #14)

1. in terminal #1, execute the command

   ```
   nc -l localhost 1234 > B.txt
   ```

   which acts as the server, *then*

2. in terminal #2, compute a MAC tag for `A.txt`

   ```
   cat A.txt | openssl dgst -sha1 -hmac 'secret' | cut -d ' ' -f 2 > T.txt
   ```

   then send `T.txt` *and* `A.txt`

   ```
   cat T.txt A.txt | nc localhost 1234
   ```

   rather than just the content in `A.txt`, *then*

3. back in terminal #3, in `B.txt` we now have the MAC tag *and* file content; first we need to separate them

   ```
   cat B.txt | head -c 41 > C.txt
   cat B.txt | tail -c +42 > D.txt
   ```

   (where 41 hexadecimal characters represents 160 bits of SHA-1 output plus a newline character), before recomputing the MAC on `D.txt` (i.e., the file content) as follows

   ```
   cat D.txt | openssl dgst -sha1 -hmac 'secret' | cut -d ' ' -f 2 > E.txt
   ```

   and finally comparing the result in `E.txt` with the MAC tag sent

   ```
   cmp C.txt E.txt
   ```

If/when you get this working, the next step is to (re)introduce the active man-in-the-middle: have it manipulate the communication from terminal #3 to terminal #1, and verify that comparing the communicated and recomputed MAC tags can detect this.

3.3 Asymmetric encryption and decryption operations

Using an *a*symmetric primitive to encrypt and decrypt data is almost as easy as with a symmetric primitive (such as the block cipher AES, which we already explored). However, before doing so, and having fetched some data as before

```
<A.txt 'http://www.gutenberg.org/dirs/etext97/1ws4110.txt'
bash$
```

we first need to generate a pair of public and private keys. When using RSA [**rsa**] for example, this is achieved as follows

```
bash$ openssl genrsa -out rsa_sk.pem 1024
Generating RSA private key, 1024 bit long modulus
.......++++++
..........................++++++
e is 65537 (0x10001)
bash$ openssl rsa -in rsa_sk.pem -pubout -out rsa_pk.pem
writing RSA key
bash$
```

with the second command extracting the public key into a separate file for clarity. We can inspect the resulting key material as follows:

```
    00:d5:ef:56:01:d4:83:e3:cf:55:ff:3c:05:49:50:
    c3:32:b1:bd:9e:57:eb:58:31:9d:2b:2e:75:42:6c:
    51:40:d0:ed:53:d0:bb:39:aa:89:6c:16:d7:a7:bb:
    dd:d4:92:a0:1a:9c:d3:db:83:b4:04:89:7f:fe:55:
    13:65:21:0f:69
exponent1:
    43:fd:1e:2f:f2:db:c4:72:02:5d:9e:9a:10:33:ce:
    44:15:69:1a:5d:ad:e0:70:e4:d0:ac:46:48:b1:3b:
    bb:6d:70:9c:55:eb:b4:e8:7b:d3:4a:ce:f3:8f:61:
```

```
    63:67:f5:f3:b9:33:3c:25:ff:1e:3a:07:10:3a:a4:
    9d:68:76:45
exponent2:
    72:fe:29:f1:e3:e4:5c:e0:86:ae:21:fa:09:75:92:
    e6:bd:e9:59:a2:92:8b:1e:68:07:dc:fa:04:91:2a:
    7f:b7:8d:c9:fe:a9:94:8e:99:3d:73:6d:c9:e8:4e:
    8d:c3:9e:b1:8f:58:c5:66:9e:ad:cd:a6:f1:f3:92:
    f0:49:fb:f9
coefficient:
    6a:c6:a2:3c:6f:59:21:08:83:5d:b2:a9:f5:8e:de:
    a8:14:85:b5:4f:6c:c0:25:89:a5:af:82:f3:0b:42:
    37:0e:ee:9f:66:5b:3b:e5:71:5a:a4:a1:d2:69:c7:
    4c:1c:62:89:fa:84:6e:4c:fd:a4:2d:88:fb:97:52:
    49:4b:3c:26
bash$
```

You need to know how RSA works to interpret the output, but based on the overview in Chapter 10 one can identify hexadecimal values of p and q, N, e and d for instance. Now we can try to encrypt `A.txt`, then decrypt the result again much like we did previously with the symmetric example:

```
bash$ openssl rsautl -encrypt -pubin -inkey rsa_pk.pem -in A.txt -out B.txt
bash$
```

There is a problem: RSA can only encrypt messages, say m, which are smaller than N. Put another way, it requires $0 \leq m < N$ but we gave it $m =$ `A.txt` which is many kilo bytes in size and hence $m > N$. In reality, we would need to split `A.txt` into suitably sized blocks. Here however, imagine we only care about the first 64B block of the file. We first extract this block into `B.txt` using `head`, then proceed as follows:

```
bash$ cat A.txt | head -c 64 > B.txt
bash$ openssl rsautl -encrypt -pubin -inkey rsa_pk.pem -in B.txt -out C.txt
bash$ openssl rsautl -decrypt        -inkey rsa_sk.pem -in C.txt -out D.txt
bash$ cat B.txt | od -Ad -tx1 -w16 -N64
0000000
bash$ cat C.txt | od -Ad -tx1 -w16 -N64
0000000 60 74 8a 8c 1f 77 46 3b b9 66 47 63 80 99 4e 35
0000016 a5 b3 c4 0c 7b 74 d7 d6 40 59 df bc 8c 70 22 36
0000032 de 70 2f 69 ac 20 10 85 72 2d 8e 03 3f 12 20 23
0000048 6a 68 77 87 39 1f b3 48 83 4c 04 c0 e6 f8 65 fa
0000064
bash$ cat D.txt | od -Ad -tx1 -w16 -N64
0000000
bash$ cmp B.txt D.txt
bash$
```

As an aside, the `split` command can be useful in circumstances such as this: it could be used to split `A.txt` into as many equally sized blocks as required (rather than just extracting the first block as above). Either way, notice the encryption step uses the public key `rsa_pub.pem`, while the decryption step uses the private key `rsa_pri.pem`; in the former, we tell OpenSSL that we are using a public key via the `-pubin` option. As in the symmetric case, use of an given input (resp. output) file can of course be replaced by use of `stdin` (resp. `stdout`) by omitting the `-in` (resp. `-out`) option.

3.4 Asymmetric signature and verification operations

As above, using an *a*symmetric digital signature algorithm instead of a symmetric MAC is fairly similar. Once we have some data

```
<A.txt 'http://www.gutenberg.org/dirs/etext97/1ws4110.txt'
bash$
```

we again need to start by generating the public and private keys. For DSA [7], a further requirement is that we generate some domain parameters (the first command) which can be viewed as global data that everyone knows:

```
bash$ openssl dsaparam -out dsa_param.pem 1024
Generating DSA parameters, 1024 bit long prime
This could take some time
....................+..+++++++++++++++++++++++++++++++++++++++++++++++++++*
......+....+.....+.......................+....+........+......................
..++++++++++++++++++++++++++++++++++++++++++++++++++*
bash$ openssl gendsa dsa_param.pem -out dsa_sk.pem
Generating DSA key, 1024 bits
bash$ openssl dsa -in dsa_sk.pem -pubout -out dsa_pk.pem
read DSA key
writing DSA key
bash$
```

In this case, the generation process is driven by randomness read from `/dev/urandom`; once the domain parameters are generated we then generate the public and private keys (second and third commands) much like RSA, again extracting the public key into a separate file for clarity. The end result can again be inspected as follows:

```
P:
    00:c2:57:aa:8b:71:23:cd:6c:dc:8e:6a:0a:76:6f:
    58:42:bd:69:50:9e:b8:ee:e5:9d:fd:34:a4:c9:a0:
    41:5a:04:3c:81:24:bd:4a:b4:7a:22:79:1c:42:5c:
    23:ba:17:e3:01:12:32:98:0a:84:eb:62:95:de:94:
    33:29:f7:4c:e8:6f:ca:2e:57:74:42:de:fa:9e:c7:
    6f:8f:0f:05:ec:14:cf:3f:0c:79:84:a4:40:84:83:
    7a:c9:7d:fa:33:f3:8f:69:4b:7a:7b:53:e7:0e:1c:
    40:ce:5e:4b:9a:da:76:f2:01:21:61:15:3d:a5:56:
    b4:5a:19:c3:9a:84:27:dc:2d
Q:
    00:d3:93:13:1c:16:e5:74:7d:d9:de:b8:05:76:32:
    24:13:25:8a:5f:9b
G:
    00:bf:02:a3:90:16:f8:d2:7e:e4:d6:c7:f7:95:82:
    c4:70:7d:65:12:41:a9:5c:cf:42:5b:9b:d4:26:d2:
    68:81:e0:fb:df:c1:97:61:fc:8c:25:04:41:ee:09:
    58:cc:06:0f:d4:bd:33:f5:94:37:b1:78:ea:e3:0e:
    5b:39:cb:21:d9:86:84:f3:3b:4d:47:c6:ae:73:a2:
    d7:ac:a6:7c:c3:92:cd:d0:39:d9:5e:07:f4:63:4a:
    ff:c7:12:43:9e:13:a1:74:68:8d:2a:69:20:5e:05:
    1e:8b:0c:ff:06:b4:74:be:cd:de:b7:7a:c1:70:75:
    dd:01:98:f5:d1:59:0c:ec:ab
bash$
```

Now we can sign the data using our private key:

```
bash$ openssl dgst -dss1 -sign   dsa_sk.pem -out       B.txt A.txt
bash$ openssl dgst -dss1 -verify dsa_pk.pem -signature B.txt A.txt
Verified OK
bash$
```

The first command above produces the signature in `B.txt`, and the second command verifies this signature on `A.txt` using our public key: in this case, the two match and so the command succeeds. What happens if we try to verify the signature on some other message, say some `C.txt` where like the MAC example we change all the 'a' characters with 'b'? As one might expect, this time the verification fails:

```
bash$ cat A.txt | tr 'a' 'b' > C.txt
bash$ openssl dgst -dss1 -verify dsa_pk.pem -signature B.txt C.txt
Verified OK
bash$
```

Of course using an asymmetric primitive to sign a large message is fairly inefficient. To improve the example therefore, we *could* first apply a hash function the message and then sign the resulting digest, i.e.,

```
<txt | openssl dgst -dss1 -sign   dsa_sk.pem -out       B.txt
<txt | openssl dgst -dss1 -verify dsa_pk.pem -signature B.txt
Verified OK
bash$
```

The same reasoning applies: if we change `A.txt` then the digest produced by SHA-1 will change, so the signature verification will fail. So provided the hash function is secure, signing the (shorter) digest produces the same result from a security perspective but is clearly more efficient.

4 Experimenting with SSL and TLS using OpenSSL

As we already mentioned, the high-level goal of OpenSSL is to support use of the SSL and TLS protocols; in a sense, everything else it offers is a side-effect of this. Put another way, SSL and TLS motivates provision of block ciphers etc. by OpenSSL since both protocols combine such primitives to achieve their high-level goal. As a result, experimenting with SSL and TLS can help illuminate more theoretical study. Specifically, we can *see* how they usefully combine the primitives we have already used by hand from the command-line.

Imagine we want to establish a secure communication channel between two hosts $\mathcal{A}$ and $\mathcal{B}$, which respectively execute processes (say **A** and **B**) representing a web-browser (or client) and web-server. Assuming the use of RSA-based signatures, we first need to generate appropriate public and private keys for the server:

```
bash$ openssl genrsa -out server.pem 1024
Generating RSA private key, 1024 bit long modulus
..........++++++
......++++++
e is 65537 (0x10001)
bash$
```

Next we need a X.509 certificate [34] so the server can authenticate itself to clients. Of course we lack a CA to issue a real certificate, so instead opt to generate a self-signed alternative as follows:

```
<dress=page@cs.bris.ac.uk' -key server.pem -out server.cert
bash$
```

This is fairly complicated, but can be roughly summarised as follows:

- the `-subj` option specifies the identity relating to $\mathcal{B}$, representing the subject of the certificate[1],
- the `-key` option specifies the file from which a public-key $PK_{\mathcal{B}}$ and a private-key $SK_{\mathcal{B}}$ are read,
- the `-out` option specifies the file into which a certificate

$$\sigma = \text{RSA.SIG}(SK_{\mathcal{B}}, (\mathcal{B}, PK_{\mathcal{B}}))$$

  is then written.

We can inspect the result, via

```
                    75:36:59:d0:2c:90:d7:f6:4a:6a:48:12:ae:6c:4c:
                    08:ae:32:00:17:aa:1a:1c:60:da:d2:57:22:81:5a:
                    24:1b:c5:93:8a:0c:92:4e:18:04:02:dd:61:a8:f7:
                    97:e4:a7:d4:1b:9c:7b:41:53
                Exponent: 65537 (0x10001)
        X509v3 extensions:
            X509v3 Subject Key Identifier:
                17:73:08:21:7A:7F:CA:5A:6A:1C:5A:3D:A9:CD:D9:BA:C1:A6:AC:A4
            X509v3 Authority Key Identifier:
                keyid:17:73:08:21:7A:7F:CA:5A:6A:1C:5A:3D:A9:CD:D9:BA:C1:A6:AC:A
4

            X509v3 Basic Constraints:
                CA:TRUE
    Signature Algorithm: sha256WithRSAEncryption
         59:a1:87:c9:5e:63:3b:39:eb:6d:36:f8:1c:c9:e1:e7:79:32:
         33:a1:bd:95:1c:93:68:02:d5:3d:b5:b9:48:7e:d2:79:01:15:
         66:3c:64:45:ba:d9:ab:85:bf:61:d9:64:7a:8e:cb:2a:b4:13:
         fa:6d:2e:58:e7:79:7c:6e:60:df:43:16:73:14:93:5b:11:08:
         bd:a0:42:5e:21:89:b8:90:8c:90:6b:3d:ea:10:ea:2b:1e:15:
         a9:bb:55:15:94:d0:ae:57:dc:ac:e5:c5:b6:67:7b:f0:1b:28:
         c8:2e:7e:46:83:f1:d8:af:45:7f:f8:37:39:ac:24:e5:b0:02:
         49:54
bash$
```

noting, for example, that the public key information help within the certificate matches that in `server.pem`. Using the key material and certificate, we are now ready to establish the secure communication channel using TLS. OpenSSL includes a variety of tools that make this very easy:

- Executing

  ```
  openssl s_server -state -accept 1234 -key server.pem -cert server.cert -msg -www
  ```

  on some host, say `foo`, launches a TLS test server, where

  - `-state` instructs the server to produce a dump of all TLS session states,
  - `-accept` specifies the port to listen for connections on, which in this case is 1234,
  - `-key` and `-cert` specify the (previously generated) server key and certificate,
  - `-msg` instructs the server to produce a hexadecimal dump of all communicated messages, and
  - `-www` means the server will act as a (very) simple web-server, which allows connections over HTTP and sends back a simple message (cf. web-page) with information about the server.

- Executing

  ```
  openssl s_client -state -connect foo:1234
  ```

  on any host (including `foo`) launches a TLS test client, where

  - `-state` instructs the server to produce a dump of all TLS session states, and
  - `-connect` specifies the port to connect to (i.e., the port on which a server is listening for connections), which in this case is 1234 on `foo`.

Now try the following:

[1] An identify can be formed from various fields; in this case "O" labels the organisation, "OU" labels the organisational unit, "L" labels the locality, and "CN" labels the common name.

Implement (task #15)

Reproduce the steps above, in both cases replacing foo with the local host whose DNS name is localhost: open two terminal windows, referred to as terminal #1 and #2, then strictly in order, *first*

1. in terminal #1, execute the test server as above, *then*
2. in terminal #2, execute the test client as above.

Once finished, terminate the test client in terminal #2. Then use a *real* web-browser such as Chrome or `links` to access the URL

https://localhost:1234/

and hence interact with the test server.

Challenge (task #16)

You already have the tools and have seen how to introduce a passive or active man-in-the-middle between a simple `nc`-based client and server. Can you reproduce such an attacker between a more complex TLS-based OpenSSL test client and server?

You should find TLS detects and/or prevents each attacker: using what you observe for example, identify the feature within TLS that does so in each case.

Challenge (task #17)

In Task 15, the information sent from the test server back to the test client or (real) web-browser included the TLS cipher suite [5] agreed: roughly speaking, this represents the set of algorithms used to secure communication. The cipher suite is agreed using a one-sided negotiation: as part of the TLS hand-shake protocol,

- the client sends a list of algorithms it supports (ordered by preference), then
- the server sends back a choice from that list.

We can control the list of algorithms sent by the OpenSSL test client using the `-cipher` option:

- Use the command

  ```
  openssl ciphers
  ```

  to produce a list of valid algorithms, or more specifically valid *combinations* of them. Given the output, do some research into the primitives each combination relates to: for example, what does `EDH-RSA-DES-CBC-SHA` mean?

- Revisit Task 15, but when executing the test client use `-cipher` to control the cipher suite agreed. By experimenting with the list, can you reason about *when* and *why* the server might make one choice over another?

References

[1] *Wikipedia: Advanced Encryption Standard (AES)*. http://en.wikipedia.org/wiki/Advanced_Encryption_Standard (see p. 16).

[2] *Wikipedia: ARPANET*. http://en.wikipedia.org/wiki/ARPANET (see p. 7).

[3] *Wikipedia: Block cipher*. http://en.wikipedia.org/wiki/Block_cipher (see p. 16).

[4] *Wikipedia: Block cipher modes of operation*. http://en.wikipedia.org/wiki/Block_cipher_modes_of_operation (see p. 16).

[5] *Wikipedia: Cipher suite*. http://en.wikipedia.org/wiki/Cipher_suite (see p. 25).

[6] *Wikipedia: Denial-of-Service attack*. http://en.wikipedia.org/wiki/Denial-of-service_attack (see p. 8).

[7] *Wikipedia: Digital Signature Algorithm (DSA)*. https://en.wikipedia.org/wiki/Digital_Signature_Algorithm (see p. 22).

[8] *Wikipedia: Domain Name System (DNS)*. http://en.wikipedia.org/wiki/Domain_Name_System (see p. 5).

[9] *Wikipedia: Ethernet*. http://en.wikipedia.org/wiki/Ethernet (see pp. 5, 10).

[10] *Wikipedia: Firewall*. http://en.wikipedia.org/wiki/Firewall_(computing) (see p. 3).

[11] *Wikipedia: Hash-based Message Authentication Code (HMAC)*. http://en.wikipedia.org/wiki/Hash-based_message_authentication_code (see p. 19).

[12] *Wikipedia: HyperText Transfer Protocol (HTTP)*. http://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol (see p. 14).

[13] *Wikipedia: Internet Control Message Protocol (ICMP)*. http://en.wikipedia.org/wiki/Internet_Control_Message_Protocol (see p. 7).

[14] *Wikipedia: Internet Protocol (IP)*. http://en.wikipedia.org/wiki/Internet_Protocol (see p. 4).

[15] *Wikipedia: Kernel*. http://en.wikipedia.org/wiki/Kernel_(computing) (see p. 3).

[16] *Wikipedia: Link-local address*. http://en.wikipedia.org/wiki/Link-local_address (see p. 10).

[17] *Wikipedia: Load balancing*. http://en.wikipedia.org/wiki/Load_balancing_(computing) (see p. 8).

[18] *Wikipedia: Loop-back*. http://en.wikipedia.org/wiki/Loopback (see p. 10).

[19] *Wikipedia: MD5*. http://en.wikipedia.org/wiki/MD5 (see p. 17).

[20] *Wikipedia: Network Interface Controller (NIC)*. http://en.wikipedia.org/wiki/Network_interface_controller (see p. 4).

[21] *Wikipedia: OSI model*. http://en.wikipedia.org/wiki/OSI_model (see p. 4).

[22] *Wikipedia: Packet Analyser*. http://en.wikipedia.org/wiki/Packet_analyzer (see p. 3).

[23] *Wikipedia: Ping flood* (see p. 8).

[24] *Wikipedia: Ping of death*. http://en.wikipedia.org/wiki/Ping_of_death (see p. 8).

[25] *Wikipedia: Port*. http://en.wikipedia.org/wiki/Port_(computer_networking) (see p. 5).

[26] *Wikipedia: Remote Procedure Call (RPC)*. http://en.wikipedia.org/wiki/Remote_procedure_call (see p. 11).

[27] *Wikipedia: Salt*. http://en.wikipedia.org/wiki/Salt_(cryptography) (see p. 17).

[28] *Wikipedia: SHA-1*. http://en.wikipedia.org/wiki/SHA-1 (see p. 19).

[29] *Wikipedia: Sub-network*. http://en.wikipedia.org/wiki/Subnetwork (see p. 4).

[30] *Wikipedia: Superuser*. http://en.wikipedia.org/wiki/Superuser (see p. 3).

[31] *Wikipedia: Talk*. http://en.wikipedia.org/wiki/Talk_(software) (see p. 13).

[32] *Wikipedia: Transmission Control Protocol (TCP)*. http://en.wikipedia.org/wiki/Transmission_Control_Protocol (see p. 4).

[33] *Wikipedia: Transport Layer Security (TLS)*. http://en.wikipedia.org/wiki/Transport_Layer_Security (see p. 16).

[34] *Wikipedia: X.509*. http://en.wikipedia.org/wiki/X.509 (see p. 23).

I have a question/comment/complaint for you. Any (positive *or* negative) feedback, experience or comment is very welcome; this helps us to improve and extend the material in the most useful way.

However, keep in mind we are far from perfect; mistakes are of course possible, although hopefully rare. Some cases are hard for us to check, and make your feedback even more valuable: for instance

1. minor variation in software versions can produce subtle differences in how some commands and hence examples work, and
2. some examples download and use online resources, but web-sites change over time (or even might differ depending on where you access them from) so might cause the example to fail.

Either way, if you spot a problem then let us know: we will try to explain and/or fix things as fast as we can!

Can I use this material for something ? We are distributing these PDFs under the Creative Commons Attribution-ShareAlike License (CC BY-SA)

http://creativecommons.org/licenses/by-sa/4.0/

This is a fancy way to say you can basically do whatever you want with them (i.e., use, copy and distribute it however you see fit) as long as a) you correctly attribute the original source, and b) you distribute the result under the same license.

Is there a printed version of this material I can buy? Yes: Springer have published selected Chapters in

http://www.springer.com/computer/book/978-3-319-04041-7

while *also* allowing us to keep electronic versions online. Any royalties from this published version are donated to the UK-based Computing At School (CAS) group, whose ongoing work can be followed at

http://www.computingatschool.org.uk/

Why are all your references to Wikipedia? Our goal is to give an easily accessible overview, so it made no sense to reference lots of research papers. There are basically two reasons why: research papers are often written in a way that makes them hard to read (even when their intellectual content is not difficult to understand), and although many research papers are available on the Internet, many are not (or have to be paid for). So although some valid criticisms of Wikipedia exist, for introductory material on Computer Science it certainly represents a good place to start.

I like programming; why do the examples include so little programming? We want to focus on interesting topics rather than the mechanics of programming. So even when we include example programs, the idea is to do so in a way where their meaning is fairly clear. For example it makes more sense to use pseudo-code algorithms or reuse existing software tools than complicate a description of something by including pages and pages of program listings.

But you need to be able to program to do Computer Science, right? Yes! But only in the same way as you need to be able to read and write to study English. Put another way, reading and writing, or grammar and vocabulary, are just tools: they simply allow us to study topics such as English literature. Computer Science is the same. Although it *is* possible to study programming as a topic in itself, we are more interested in what can be achieved *using* programs: we treat programming itself as another tool.