Email 101

David F. Skoll

March 16, 2010

1 Introduction

The flow of email on the Internet is both surprisingly simple and surprisingly complex. It is simple because the basic email protocols are decades-old plain-text protocols with very simple definitions. It is complex because many tiny details can have a profound influence on how mail flows and on how to interpret message headers.

This document will teach you how email flows on the Internet, and how to interpret message headers to determine the history of a piece of email. These are valuable skills for troubleshooting your email setup or even just for figuring out how a particular email arrived in your inbox.

2 Protocols

A *protocol* is simply an agreed-upon way for two pieces of software or hardware to communicate. It's the basic set of ground rules to ensure the two ends understand the conversation that's going on between them.

Protocols are typically developed in many layers, and a whole layered set of protocols is sometimes called a *protocol stack*. For example, two computers on a LAN use protocols such as these:

- At the lowest level, the Ethernet physical protocol defines the voltage levels and timing on the Ethernet cable.
- At a slightly higher level, the Ethernet data link protocol defines the bit patterns that make up a valid Ethernet frame.
- At higher levels are network, transport and application protocols, which will be discussed later.

In this document, we won't care about the very low-level protocols such as Ethernet physical and data link protocols. We will start our look at protocols at the network layer and move up.

2.1 RFCs

The Internet protocols are standardized in a series of documents called *RFCs*. RFC stands for *Request for Comments*. An organization called the *IETF* (Internet Engineering Task Force) is responsible for managing RFCs. The RFCs have some unusual features:

1. They are completely free and available online. Many standards organizations make quite a lucrative business out of charging for their standards documents. This is not the case for the IETF.
2. RFCs tend to be written by one or a few authors. As a result, they are usually quite clear. They also tend to avoid "standards-by-committee" syndrome—they are much simpler than standards documents from typical organizations like the ISO (International Organization for Standards) and ITU-T (Telecommunication Standardizations Sector of the International Telecommunication Union.)
3. RFCs are available in plain-text format. They are easy to read online on any type of computer.

3 TCP/IP

The protocol stack that runs the Internet is called TCP/IP, which stands for *Transmission Control Protocol / Internet Protocol*. It is the protocol stack that has been running the Internet for several decades.

Technically, the current IP protocol is IP version 4, known as *IPv4*. A new protocol known as *IPv6* fixes some shortcomings in IPv4 and will eventually replace it. However, for our purposes, the differences between IPv4 and IPv6 are small, so this document will simply talk about the "IP" protocol without distinguishing between IPv4 and IPv6 except where noted.

3.1 The Network Layer

The network layer in TCP/IP is called *IP*, the *Internet Protocol*. IP's job is to take a chunk of data (called a *packet*) from one machine, and deliver it to another machine.

In IP, each network interface (such as an Ethernet card) is assigned an *IP Address*. This is simply a 32-bit number (128 bits in IPv6) that identifies the interface. IPv4 addresses are often written as four decimal numbers separated by dots, like this: `192.168.10.3`.

IP's job is to take a packet from one IP address called the *source* and do its best to make sure the packet gets to another IP address called the *destination*. What happens internally in the network is quite complex, but the end result is that when you hand off a packet to IP, it (probably) gets to the destination in a short time.

IP has the following characteristics:

- Its job is to get packets from one IP address to another.
- IP does not make guarantees that the packet will arrive. If you send several IP packets, any of the following may happen:
  - The packets could all arrive, in order.
  - The packets could arrive, but out of order.
  - Some packets might never arrive.
  - Some packets might be duplicated: The destination address might see extra copies.

  Of course, most of the time, an IP packet arrives just fine, without duplicates. But IP does not *guarantee* this; it's up to higher layers to deal with lost or duplicated packets.

Figure 1 summarizes the operation of IP:

Figure 1: IP

3.2 The Transport Layer

The transport layer builds on the network layer to allow communication between two *programs*. It may optionally also add reliability to the network layer.

Whereas the network layer is just concerned with getting packets from one IP address to another, the transport layer discriminates between different programs running on the same IP address. For example, a single machine could run both a mail server and a Web server. When you connect to the machine, you want to make sure that your Web browser talks to the Web server, while your email client talks to the mail server.

In TCP/IP, different endpoints at the same IP address are distinguished by having different *port numbers*. A port number is simply a 16-bit integer from 0 to 65535. If you think of an IP address as a business phone number, a port is like an extension. The phone number reaches the business and the extension reaches a particular person. Similarly, an IP address reaches a machine and a port number reaches a particular program running on that machine.

TCP/IP has two different transport layers: UDP and TCP.

3.2.1 UDP

UDP, which stands for *User Datagram Protocol*, is a very simple layer built on top of IP. About all it adds are port numbers. It doesn't add any reliability. So UDP will take a packet of data from one program and make a best-effort attempt to get it to another program. All of IP's caveats about packets going missing or being duplicated apply to UDP; an application that uses UDP must be prepared to handle missing or duplicated packets.

3.2.2 TCP

TCP, which stands for *Transmission Control Protocol*, is a much more complicated layer built on top of IP. In addition to adding port numbers, TCP gives two programs the illusion that they have a dedicated connection between them.

Part of TCP's job is to solve the problems of missing and duplicated data. Any data one program puts into the TCP connection will be seen by the other end. The data will come out in exactly the order it went in, and there will not be any duplicates.

Although TCP is very complicated internally, with a sophisticated system of timers and retransmissions to work around IP's unreliability, we can view a TCP connection as a two-way pipe: Whatever sequence of bytes a program puts into one end, the same sequence comes out reliably on the other end.

Figure 2 illustrates the operation of TCP:

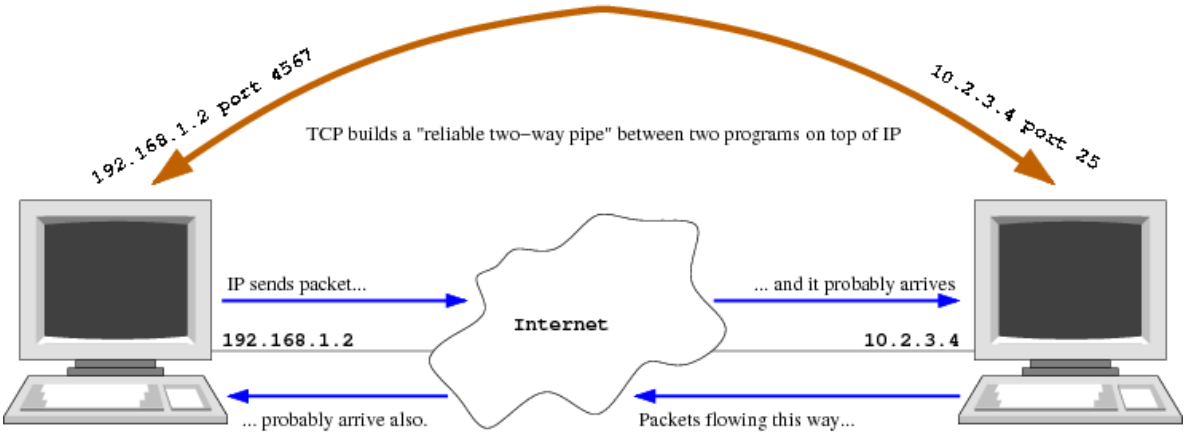

Figure 2: TCP

Usually, the program initiating a TCP connection is called a *client* and the program on the other end that accepts the connection is called a *server*. The server is said to *listen* for clients as it awaits new connections.

Figure 2 shows a client (on the left) using port 4567 while the server uses port 25. A novice mistake is to misunderstand the client's port when creating firewall rules. An SMTP server will always listen on well-known port 25 while a client will connect from any unreserved port number.

4 The Application Layer

So far, we have looked at protocol layers whose job is to move data from one place to another. The *application layer* is the first layer that actually gives the data *meaning*, rather than treating it as a bunch of bits to move around. We'll look at several TCP/IP applications and how they relate to email delivery.

Many applications have servers that listen on specific ports. These ports are reserved for use by those applications, and are called *well-known ports*.

4.1 DNS

The DNS, or *Domain Name System*, is a system for translating names into IP addresses. (It has many other uses, but this is its primary use.)

TCP/IP internally knows computers only by their IP addresses. IP, for example, knows nothing about `www.roaringpenguin.com`. It only knows about `70.38.112.54`.

Since people are better at remembering names than numbers, most people give their computers names. The DNS translates from human-friendly names to network-friendly IP addresses.

In the old days when the Internet was the ARPANet, this translation was done by shipping around a file listing all host names on the network along with their corresponding IP addresses. This worked fine when the network was small, but is completely impractical on today's Internet.

The DNS is a *large distributed database*. It is *large* because it contains hundreds of millions of entries mapping host names to IP addresses (as well as other kinds of entries we'll discuss later.) It is *distributed* because no one computer holds the entire DNS database. Instead, millions of computers across the world hold small portions of the database. A computer that replies to DNS requests is called a *name server*.

In the DNS, each *domain name* consists of a series of labels separated by dots. The rightmost label is called the *top-level domain*. Top-level domains include the familiar standbys like `com`, `net`, `org` and so on, as well as *country-code* domains like `us`, `ca`, `fr` and so on. Several other top-level domains such as `biz`, `aero` and `museum` exist, but they haven't really caught on.

A full machine name might be something like `www.roaringpenguin.com` or `www.bbc.co.uk`. Such a name is called a *domain name*. (Even though the name corresponds to a *host*, the DNS still refers to it as a domain name.)

To translate a name like `www.roaringpenguin.com` into an IP address, the DNS *client* performs the following actions:

1. It asks one of the *root name servers* for the machines that know about the `com` top-level domain. The root name servers are a set of machines distributed around the world whose IP addresses are well-known. These IP addresses are the starting point for doing lookups; they must be known ahead of time to avoid a chicken-and-egg problem.
2. The root name server replies with a list of name servers for the `com` domain.
3. The DNS client next asks one of the `com` name servers for the machines that know about the `roaringpenguin.com` domain.
4. The `com` name server replies with a list of name servers for `roaringpenguin.com`
5. The DNS client asks one of the `roaringpenguin.com` name servers for the IP address of `www.roaringpenguin.com`.
6. The `roaringpenguin.com` name server replies with the IP address.
7. The DNS client *caches* the result. That is, it remembers the answer for a certain period of time. If someone else asks for the IP address of `www.roaringpenguin.com` while the answer is cached, the DNS client can reply immediately without having to redo all of the lookups. The caching of DNS replies is crucial to making the DNS scalable and efficient. Without caching, the load on the root name servers would quickly bring them to their knees.

Figure 3 illustrates how the address of `www.roaringpenguin.com` might be looked up. Questions are shown in red and replies are shown in green.

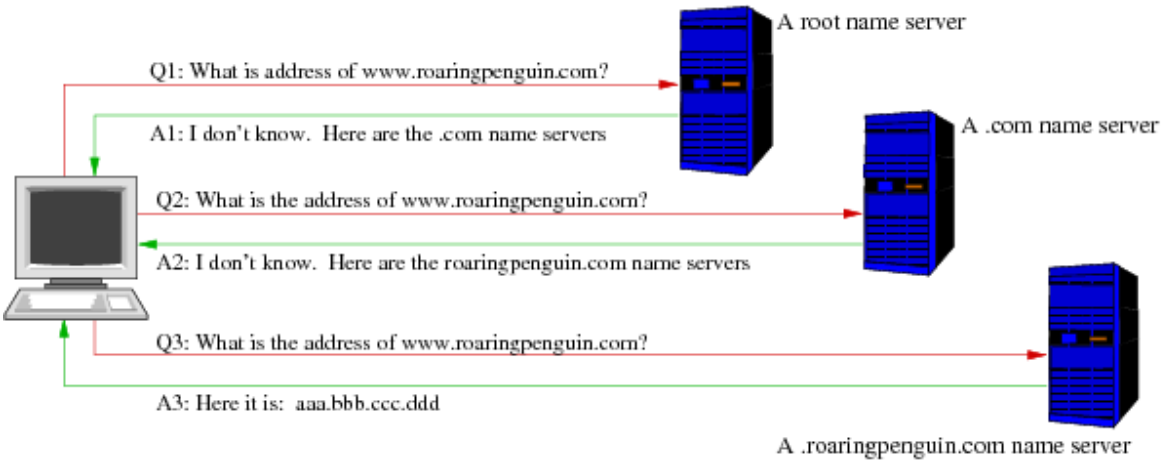

Figure 3: DNS Lookup Example

4.1.1 DNS Transport

The DNS usually uses UDP for requests and responses. However, a large query or response might cause it to switch to TCP. A common novice mistake is to set up a firewall to allow DNS queries over UDP, but block them over TCP. This can result in hard-to-diagnose failures; be sure your firewall is set up to allow DNS queries over both UDP and TCP.

DNS always uses the well-known port 53, both for UDP and TCP.

4.1.2 DNS Record Types

In addition to returning an IP address given a domain name (a so-called **A** record), the DNS contains other *record types*:

- A **PTR** record contains the reverse information from an A record. That is, given an IP address, we can determine the domain name associated with that IP address. (This is a so-called *reverse lookup*; not all sites set up reverse lookups.)
- An **MX** record contains information about how to route mail for a domain. A given domain can have multiple MX records, meaning there are multiple machines willing to accept mail for the domain. Each MX record contains a host name and a *cost* (sometimes referred to as a *preference*.) When sending mail to a domain, the MX records are tried in increasing order of cost. If two MX records have the same cost, they are supposed to be tried in random order.

There are many other DNS record types, but for the purposes of email delivery, the most important are **A** and **MX** records.

4.2 DNS Lookup Examples

If you have access to a UNIX or Linux machine with the `host` program installed, you can do some DNS lookups at the command prompt. Here are some examples:

```
#...Look up the A record of www.sun.com
$ host www.sun.com
www.sun.com has address 72.5.124.61

#...Lookup up the MX records of sun.com
$ host -t mx sun.com
sun.com mail is handled by 5 btmx4.sun.com.
sun.com mail is handled by 5 btmx6.sun.com.
sun.com mail is handled by 20 mx3.sun.com.
sun.com mail is handled by 20 mx4.sun.com.

#...Do a reverse-lookup on 70.38.112.54
$ host 70.38.112.54
54.112.38.70.in-addr.arpa domain name pointer
roaringpenguin.com.
```

5 SMTP

SMTP stands for *Simple Mail Transfer Protocol* and is the protocol used to deliver email on the Internet. SMTP is quite old; it was first standardized in 1982 in RFC 821. Although the standard has been updated several times since then (most recently in RFC 5321), the basic flavor has not changed. Because most of this document will be devoted to discussing SMTP, we've made it a top-level section even though it logically fits under Section 4, The Application Layer.

SMTP has *absolutely no authentication*. This is something that often takes newcomers aback, but the fact is that *almost anything* in an SMTP session can be faked: *You can't trust the sender's address or most of the message headers and body.*

SMTP is a simple text-based protocol that runs over TCP. A machine wishing to send email (the SMTP *client*) connects over TCP to port 25 on the SMTP *server*. The client issues SMTP *commands* and the server responds with SMTP *replies*. All commands and replies are in plain-text, human-readable format.

The back-and-forth command/reply sequence is called an *SMTP conversation*. To illustrate the flavor of SMTP, we show an SMTP conversation below. The `C:` tag indicates data sent

by the client (the machine wishing to send mail) and the `S:` tag indicates responses sent by the server. The tags and line numbers themselves are not actually part of the conversation.

```
1  C: (Connects to server on TCP port 25.)
2  S: 220 vanadium.roaringpenguin.com ESMTP
3  C: HELO hydrogen.roaringpenguin.com
4  S: 250 vanadium.roaringpenguin.com Pleased to meet you
5  C: MAIL FROM:<dfs@roaringpenguin.com>
6  S: 250 2.1.0 <dfs@roaringpenguin.com>...  Sender ok
7  C: RCPT TO:<devnull@roaringpenguin.com>
8  S: 250 2.1.5 <devnull@roaringpenguin.com>...  Recipient ok
9  C: RCPT TO:<noone@roaringpenguin.com>
10 S: 550 5.1.1 <noone@roaringpenguin.com>...  User unknown
11 C: DATA
12 S: 354 Enter mail, end with "." on a line by itself
13 C: (Transmits message body)
14 C: .
15 S: 250 2.0.0 nBTI6n72025476 Message accepted for delivery
16 C: QUIT
17 S: 221 2.0.0 vanadium.roaringpenguin.com closing connection
18 S: (Closes connection)
```

Here is how this SMTP conversation worked:

- In Line 1, the client connected to the server on TCP port 25. Port 25 is the well-known port for SMTP.
- In Line 2, the server replied with a single line. The server's reply always begins with a three-digit *reply code* (in this case, 220.) A reply code beginning with a "2" indicates that the command was processed successfully.
- In Line 3, the client issued the HELO command followed by the client's host name. The server responded in Line 4 with a success code.
- In Line 5, the client issued the MAIL command. The email address `dfs@roaringpenguin.com` is the sender of the message. (In fact, it is the *envelope sender*; the message envelope will be discussed later.) In Line 6, the server responded with a success code.
- In Line 7, the client issued a RCPT command, indicating an intended recipient of the email. The server again responded with a success code.

- In Line 9, the client issued another RCPT command, indicating a second intended recipient. In Line 10, we see something different: The server responded with a failure code. Codes that begin with "5" indicate a permanent failure condition.
- In Line 11, the client issued a DATA command, indicating that the email body is about to follow. The server replied in Line 12 with a *conditionally successful* code. Codes beginning with "3" indicate that the preceding command was accepted, but more information is required to definitively indicate a success or failure.
- In Line 13, the client transmitted the email message, and in Line 14, it sent a single dot on a line by itself, indicating the end of the message.
- In Line 15, the server sent a successful reply code, indicating successful message delivery. (Successful only to `devnull@roaringpenguin.com`, of course.)
- In Line 16, the client issued a QUIT command indicating the end of the SMTP session. The server replied with a success code in Line 17 and then closed the connection.

The preceding SMTP conversation is typical. While there are a few other SMTP commands, almost all SMTP conversations proceed like this:

1. The client connects.
2. The client issues a HELO command as a greeting.
3. The client issues a MAIL command to indicate the sender.
4. The client issues one or more RCPT commands to indicate recipients.
5. The client issues a DATA command and transmits the message body.
6. If the client has additional messages for the server, it can reuse the connection by issuing a RSET command and going back to Step 3.
7. When the client has no more messages, it issues a QUIT command and the server closes the connection.

5.1 Email Message Data

In Section 5, we glossed over what an email message actually looks like; Line 13 just read "(Transmits message body)".

The format of an email message is specified in RFC 5322 and some other associated RFCs. An email message consists of:

- A number of plain-text *headers*.

- A blank line.
- A sequence of lines containing the message *body*.

Note that the division of an email message into headers and body is done for the convenience of email-reading software. An SMTP server (for the most part) sees both the headers and body as one big chunk of data.

A message header consists of the header *name*, a colon, and the header *value*. Some headers have strict rules setting out possible legal values, while other header values can be free-form text. Here are some examples of message headers:

```
From:  David F. Skoll <dfs@roaringpenguin.com>
To:  <devnull@roaringpenguin.com>
Subject:  Some sample message headers
```

The following are the most commonly-used message headers:

Subject	The message subject.
From	The message sender.
To	The primary message recipients.
Cc	Additional message recipients.
Date	The date the message was sent.
Message-ID	A unique identifier. No two different messages should ever have the same Message-ID.
References	Message-IDs of related messages.
In-Reply-To	The Message-ID of the message eliciting this reply.

5.2 Envelope vs. Headers

In Section 5, we saw an SMTP conversation that specified the sender email address with a MAIL command and recipient addresses with a series of RCPT commands. Then in Section 5.1, we saw the sender and recipients specified in `From:`, `To:` and `Cc:` headers.

The information contained in the MAIL and RCPT commands is called the *message envelope*. The MAIL address is called the *envelope sender* and the RCPT addresses are called the *envelope recipients*.

The address in the `From:` header is called the *header sender* and the addresses in the `To:` and `Cc:` headers are called the *header recipients*.

While there is usually a correspondence between envelope and header addresses, *there need not be any correspondence at all*. In particular, you *cannot trust any header addresses or the envelope sender.* (There's no point in faking the envelope *recipients*, because those addresses control who receives the message.)

One unfortunate side-effect of the split between envelope and header senders is this: Some technologies (such as SPF: *Sender Policy Framework*) attempt to validate the envelope sender.

However, most mail readers show the *header* sender, not the *envelope* sender. So an attacker can easily fake the `From:` line without running afoul of SPF.

5.3 SMTP Reply Codes

SMTP has four types of reply codes (we've seen three of them so far). All SMTP reply codes consist of three decimal digits, but the basic class of reply is determined by the first digit. The four classes are:

- **2***YZ*: A code beginning with 2 indicates that the command was successfully completed.
- **3***YZ*: A code beginning with 3 indicates that the command has been accepted, but further action requires additional information. We saw this with the DATA command; the SMTP server cannot really say for sure that the DATA command has succeeded until the message has been transmitted.
- **4***YZ*: A code beginning with 4 indicates that the command failed, but the failure condition was temporary. Temporary failures could be caused by a variety of conditions, such as a full disk, lack of memory, etc. Typically, the SMTP client will queue the message and try again in a little while.
- **5***YZ*: A code beginning with 5 indicates that the command failed, and the failure is likely to be permanent. (For example, sending to a nonexistent recipient could elicit a permanent failure.) The SMTP client should not queue and retry the message; instead, it should report failure to the originator of the message.

5.4 ESMTP

Since the original description of SMTP, various people and organizations have added capabilities to SMTP. In order to manage extensions to SMTP in such a way that both old and new implementations remain inter-operable, a framework for SMTP extensions called *ESMTP* (Extended SMTP) was developed.

A server that supports ESMTP must include the word "ESMTP" in its initial banner when a client connects. This alerts the client that the server supports ESMTP.

A client wishing to use ESMTP issues the command **EHLO** rather than **HELO** when sending its initial greeting. This alerts the server to the fact that the client wishes to use ESMTP.

If, for some reason, a client uses the EHLO command when talking to a server that does *not* support ESMTP, the client will get an error reply. In this case, it should issue a RSET command followed by HELO, and can then continue with a normal (non-ESMTP) SMTP dialog.

There are several ESMTP extensions, but none of them is important for this basic introduction. The only aspect of ESMTP that is interesting to us is the use of multi-line reply codes.

In normal SMTP, each reply consists of a three-digit number, a space, and some text, all on a single line. In ESMTP, multi-line reply codes can be issued. All but the last line must contain the three-digit reply code, a dash, and some text. Only the last line should contain a space instead of a dash; that is how the client knows it is the last line. Here is a snippet of a typical ESMTP conversation:

```
1   C: (Connects to server on TCP port 25.)
2   S: 220 vanadium.roaringpenguin.com ESMTP
3   C: EHLO hydrogen.roaringpenguin.com
4   S: 250-vanadium.roaringpenguin.com Pleased to meet you
5   S: 250-ENHANCEDSTATUSCODES
6   S: 250-PIPELINING
7   S: 250-EXPN
8   S: 250-VERB
9   S: 250-8BITMIME
10  S: 250-SIZE
11  S: 250-DSN
12  S: 250 HELP
```

Lines 4 through 12 are the server's multi-line response. It responded to the EHLO with a list of the ESMTP extensions it supports. The client is only allowed to use those extensions that the server advertises. Note that line 12 has a space after the 250, indicating the last line in the reply.

6 The Internet Message Format

Section 5.1 gave a basic overview of the format of an Internet message. Let's look at a message in more detail.

As mentioned in Section 5.1, an email message consists of a set of *headers* followed by a blank line and then the message *body*.

6.1 Message Headers

There are many headers besides the ones mentioned in Section 5.1. In addition, email software can add headers beginning with `X-` and use them for any purposes it wishes. With the proliferation of message headers, most email-reading software suppresses all but an "interesting" subset of headers. Unfortunately, this can sometimes hide information that can be very useful in tracing the origin of a message. You should familiarize yourself with your email reading software and learn how to get it to display *all* message headers when you need to see them.

There are two headers that are critically important for tracing the origin of an email: **Return-Path** and **Received**. Unfortunately, most email readers hide these headers! Take a moment now to learn how to convince your email-reading software to reveal the headers. Instructions for many popular programs are at http://spamcop.net/fom-serve/cache/19.html.

6.1.1 Return-Path

Return-Path header contains the *envelope* sender address. This can be very different from the **From** header. For example, consider this typical mailing list message:

```
Return-Path: <mimedefang-bounces@lists.roaringpenguin.com>
From: David F. Skoll <dfs@roaringpenguin.com>
```

As we see, the **From** header contains the email address of the person who originally sent the message. The **Return-Path** header contains the email address of the entity *actually* doing the mailing (in this case, a mailing-list program.)

Note that both the envelope sender (**Return-Path**) and **From** header can be faked. However, the **Return-Path** header will often give clues about spam email. Consider the following example:

```
Return-Path: <www-data@ns1.example.org>
From: Madam Erlisa Paradina <madam_erlisa@example.net>
Subject: Anticipate Your Urgent Response
```

Most mail-reading programs display the sender address as `madam_erlisa@example.net`. However, many web servers run as the UNIX user `www-data`. It is very likely this spam email was sent using a compromised Web server or a vulnerable web-form submission program. Although the spammer was able to change the header sender, he or she could not change the envelope sender. You can use the envelope sender (in this case) to complain to the appropriate organization about the unsecured server.

Here is another example:

```
Return-Path: <root@compromised-server.example.it>
From: Wells Fargo Online <onlineservice@wellsfargo.com>
Subject: Wells Fargo Bank -Your Online Access has been
blocked
```

This *phishing attempt* (an attempt to steal online credentials) claims to come from onlineservice@wellsfargo.com, but the envelope sender is root@compromised-server.example.it. This is particularly annoying because the wellsfargo.com domain uses Sender Policy Framework to specify which machines may send mail on its behalf. However, Sender Policy Framework would *not* have helped here because it only looks at the envelope sender, not the `From` header.

In addition, the hiding of `Return-Path` by most email readers makes it quite likely that this attempt would slip by undetected by most unsophisticated users.

In summary: If you receive an email about which you have even the slightest suspicion, take a look at `Return-Path`.

6.1.2 Received

The **Received** header is a *trace header*. That is, it is designed to let you trace the path of an email from its originator to its destination. Every SMTP server that relays an email into, within, or out of the Internet environment is supposed to prepend a **Received** header. Reading the **Received** headers from top to bottom, therefore, shows the reverse path of the email (starting from the destination and moving back to the source.)

Take a look at the **Received** headers from a sample email. (The numbers before **Received** don't appear in the email; they're just for reference.)

```
1. Received: from smtpo0.dc-uoit.net (smtpo0.dc-uoit.net [205.211.180.195])
    by colo3.roaringpenguin.com (8.14.3/8.14.3/Debian-5)
    with ESMTP id o09KGVpK013081
    for <dfs@roaringpenguin.com>; Sat, 9 Jan 2010 15:16:32 -0500

2. Received: from itx01.oncampus.local (itx01.oncampus.local [10.100.201.61])
    by smtpo0.dc-uoit.net (8.14.3/8.14.3/Debian-5)
    with ESMTP id o09KGSVk021888
    for <dfs@roaringpenguin.com>; Sat, 9 Jan 2010 15:16:28 -0500

3. Received: from itomgwpp01.oncampus.local ([10.120.201.122])
    by itx01.oncampus.local
    with Microsoft SMTPSVC(5.0.2195.6713); Sat, 9 Jan 2010 15:16:05 -0500

4. Received: from itomgwpp02.oncampus.local (10.120.201.123)
    by itomgwpp01.oncampus.local (10.120.201.122)
    with Microsoft SMTP Server (TLS) id 8.1.358.0; Sat, 9 Jan 2010 15:16:05 -0500

5. Received: from ITOMSCPP01.oncampus.local ([10.120.201.120])
    by itomgwpp02.oncampus.local ([10.120.201.123])
    with mapi; Sat, 9 Jan 2010 15:16:04 -0500
```

Starting from **Received** header 5 and working our way back, we see:

- (5) The message originated from a machine that calls itself `ITOMSCPP01.oncampus.local`, which sent the message to `itomgwpp02.oncampus.local`. The protocol used was `mapi`, which is a proprietary Microsoft protocol. The message was received by `itomgwpp02` on 2010-01-09 at 15:16:04 Eastern time (-0500 means 5 hours behind UTC.)
- (4) One second later, the message was sent from `itomgwpp02.oncampus.local` to `itomgwpp01.oncampus.local`.

- (3) Almost immediately, the message went from `itomgwpp01.oncampus.local` to `itx01.oncampus.local`.
- (2) 23 seconds later, the message went from `itx01.oncampus.local` to `smtpo0.dc-uoit.net`. The protocol in this case was ESMTP.
- (1) Finally, 4 seconds later, the message arrived at the final destination `colo3.roaringpenguin.com` from `smtpo0.dc-uoit.net`

Note that while **Received** headers can give you valuable tracing information, they can also be forged. Generally, you can trust any **Received** header added by a machine you own or that your ISP owns. Beyond that, treat them with healthy skepticism.

The general format of a **Received** header is something like this:

```
Received:  from claimed_name (resolved_name [ip_address])
          by receiving_machine (comment)
          with protocol additional_info
          for recipient; Weekday, Day Month Year Time Zone
```

Here is a description of the various parts of the header. Note that some Received headers may be missing some of the parts.

- *claimed_name* is the name that the SMTP client used in its HELO command. It can't be trusted at all.
- *resolved_name* is the name determined from a reverse-DNS lookup on the SMTP client's IP address. It's not very trustworthy.
- *ip_address* is the IP address of the SMTP client. If you trust the machine that added the Received: header, then *ip_address* is very trustworthy.
- *protocol* is the protocol used to receive the email. It is typically ESMTP or SMTP, but could be HTTP, HTTPS, mapi, or just about anything.
- `recipient` is the envelope recipient of the email. This section is often omitted as it could reveal Bcc: recipients.
- *Weekday* through *Zone* specify the date and time at which the email was processed by the machine adding the Received: header.

6.2 Message Bodies

The body of a message consists simply of a sequence of plain-text lines. Normally, message bodies have no particular structure. However, a series of standards called MIME (Multipurpose Internet Message Extensions) specifies techniques for encoding non-plain-text parts in

a message. This allows you to include attachments, images, etc. in an email. Although email clients present messages with attachments and rich media, for transmission over SMTP the messages must be encoded as plain-text MIME messages. The receiving email client decodes the MIME message to recover the attachments and rich media.

7 Summary

The Internet is built on a layered set of protocols. The lowest layers control the electrical specifications for transmitting signals and the details of each data link. The network layer is the first layer concerned with end-to-end routing of data. The transport layer enhances the network layer to allow two processes to establish communication, and optionally provides an error-free reliable communication channel. Finally, the application layer consists of those protocols used by applications—in the case of email delivery, the most prevalent protocol is SMTP.

The Domain Name System (DNS) provides a distributed database for converting names to IP addresses. It uses caching to improve performance. There are different types of DNS records; two that control mail delivery are the Mail Exchange (MX) and Address (A) records. The MX record names hosts that will accept mail for a specific domain, and A records provide the IP addresses of named hosts.

SMTP, the Simple Mail Transfer Protocol, is used to send email. It provides no authentication and any aspect of an email message can be spoofed.

Internet messages consist of plain-text headers followed by a blank line and then a plain-text body (although non-plain-text parts may be encoded using MIME.)