

Lecture 31: Filtering Out Spam

Lecture Notes on “Computer and Network Security”

by Avi Kak (kak@purdue.edu)

April 4, 2024

5:47pm

©2024 Avinash Kak, Purdue University



Goals:

- Spam and computer security
- How I read my email
- The acronyms MTA, MSA, MDA, MUA, etc.
- Structure of email messages
- How spammers alter email headers
- A very brief introduction to regular expressions
- An overview of procmail based spam filtering
- Writing Procmail recipes

CONTENTS

	<i>Section Title</i>	<i>Page</i>
31.1	Spam and Computer Security	3
31.2	How I Read My Email	6
31.3	Structure of an Email Message	14
31.4	How Spammers Alter the Email Headers — A Case Study	22
31.5	SPF, DKIM, and DMARC — The Three Important Email Authentication Protocols	26
31.6	Spam Despite SPF, DKIM, and DMARC — A Case Study	38
31.7	A Very Brief Introduction to Regular Expressions for Spam Filtering	42
31.8	Using Procmail for Spam Filtering	61
31.9	Homework Problems	80

[Back to TOC](#)

31.1 SPAM AND COMPUTER SECURITY

- Spam is a major source of malware that infects individual computers and, sometimes, entire networks.
- Much spam tries to lure you into clicking on URLs of websites that serve as hosts for viruses, worms, and trojans. Consequences of inadvertently downloading such software into your computer can be deadly — as previously described in Lecture 30.
- In addition to the dangerous spam that may try to steal information from your computer or turn it into a spambot for spreading even more spam, there is also another kind of spam these days: This consists of email generated by legitimate businesses and organizations that you either have no interest in reading or have no time for following up on. [For example, half of my spam consists of unsolicited messages sent to me by marketing companies, public relations houses, government agencies, university departments advertising their activities, and students in various parts of the world seeking to come to Purdue. Even just opening all of these messages would consume a significant portion of each day.]
- I am not much of a believer in spam filters that carry out a

statistical analysis of email to decide whether or not it is spam. These filters are also sometimes called Bayesian filters for blocking spam. A statistical filter with sufficiently low “falses” to suit my tastes would require too many samples of a certain type of spam before blocking such messages in the future. On the other hand, **with a regular-expression based filter**, once you see a spam message that has leaked through, it is not that difficult to figure out variations on that message that the spammers may use in the future. In many cases, you can design a short regular expression to block the email you just saw and all its variations that the spammer may use in the future in just one single step.

- Based on my personal experience, and in line with my above stated observation, you can design nearly 100% effective spam filters with tools that carry out regular-expression based processing of email messages. [A spam filter is close to 100% effective if it traps close to 100% of what **YOU** consider to be spam and lets through close to 100% of the messages that **YOU** consider legitimate.]
- Spam filter that are close to 100% effective for **your** specific needs in the sense defined above can only be built slowly. My spam filter has evolved over several years. It needs to be tweaked up every once in a while as the spammers discover new ways of delivering their unwelcome goods.

- Before ending this section, I'd urge you to scan through the following well-written report that was issued by CISCO in June 2019 about how you and your organization can be seriously harmed by email borne malware:

<https://blogs.cisco.com/security/threat-report-email-attacks>

Even though the report is now roughly five years old, the point made in it are just as valid today as they were then.

[Back to TOC](#)

31.2 HOW I READ MY EMAIL

- These days most folks read their email through web based mail clients. If you are at Purdue, in all likelihood, you log into Purdue's webmail service to check your email. Or, perhaps, you have it forwarded to your email account at a third party service such as that provided by gmail or yahoomail. **This way of reading email is obviously convenient for, say, English majors. However, if you happen to be a CS or a CompE major, that is not the way to receive and send your email.**
- The web based email tools can only filter out standard spam — this is, the usual spam about fake drugs, about how you can enlarge certain parts of your body, and things of that sort. But nowadays there is another kind of spam that is just as much of a nuisance. As mentioned in the previous section, you have generally well-meaning folks (and organizations) who want to keep you informed of all the great stuff they are engaged in and why you should check out their latest doings. These include local businesses, marketing companies, PR folks, etc. **When you write your own spam filter, you can deal with such email in a much more selective manner than would otherwise be the case.**
- Writing your own spam filter is also a great way to become

more proficient with regular-expression based processing of textual data.

- Shown in Figure 1 is how I receive my email.
- To understand the flow of email in Figure 1, you need to become familiar with the acronyms **MTA**, **MDA**, **MUA**, etc.
- An **MTA** (Mail Transfer Agent) is used to transfer email to another MTA in the internet. [It is also called a “Mail Transport Agent,” or a “Mail Server.” In the context of DNS, it is referred to as a “Mail Exchange Server,” as you saw in Lecture 17. Although the main function of an MTA is to exchange email with another MTA, they can also be programmed to receive email directly from MUAs and to send messages directly to the same. More generally, the client email first goes to an MSA (Mail Submission Agent) and the MSA forwards it to the MTA. By the same token, when an MTA receives email for clients in its own domain, it generally forwards the email to an **MDA** (Mail Delivery Agent) and it is the MDA’s job to send that email to the clients. However, an MTA can also be programmed to send email directly to the clients.] Let’s say someone in some corner of the world wants to send an email to **kak@purdue.edu**. As you should know from Lecture 17, the name resolver associated with the email client being used by the sender will ask the DNS servers for the IP address of the host that is designated to be the mail exchange server for the **purdue.edu** domain. Subsequently, the MTA program running on this host at Purdue will receive the email sent to me. The most popular program that is used as an MTA is known as

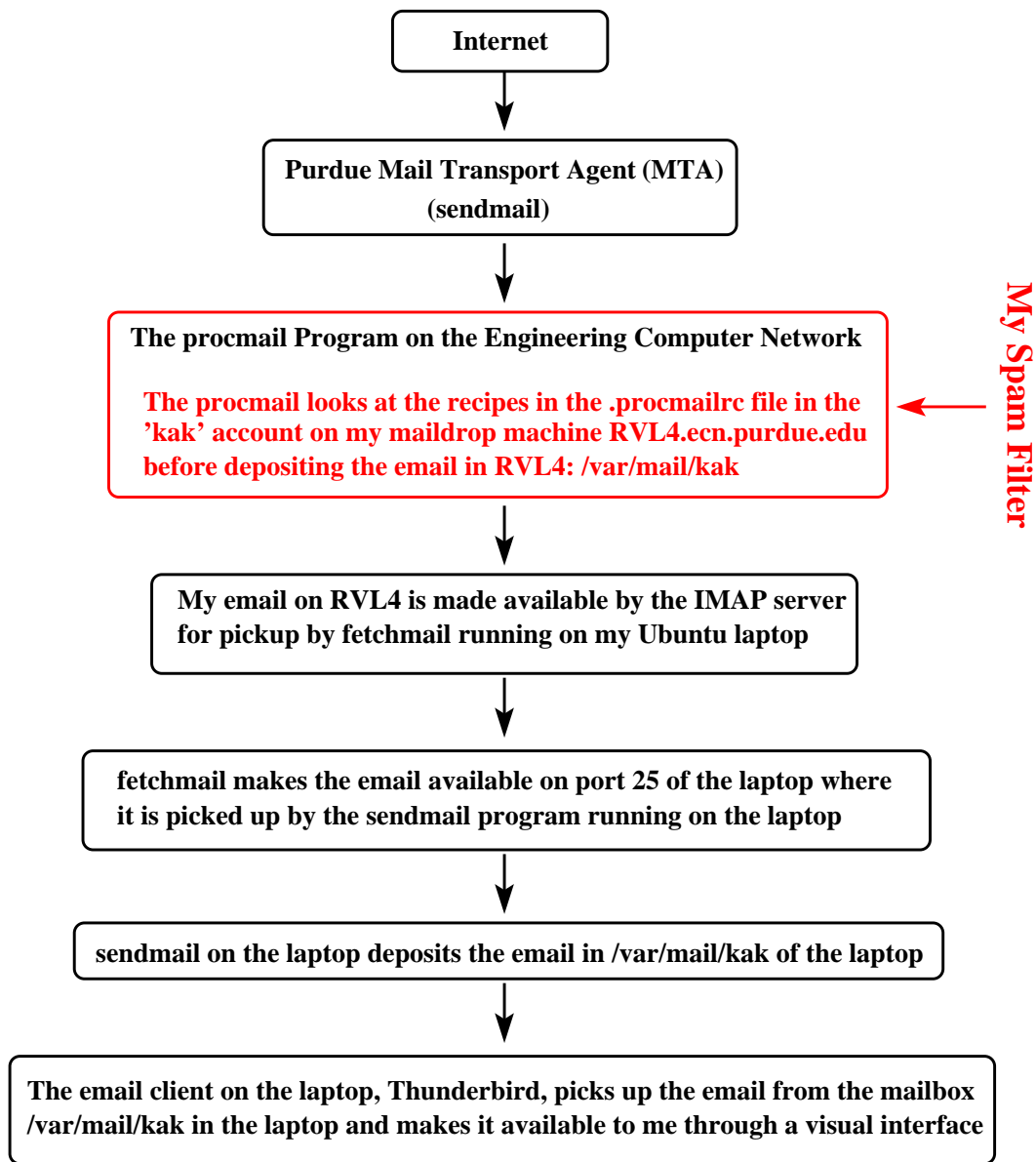


Figure 1: *This figure shows how I receive my email in my Linux laptop. The fetchmail program in my laptop picks up my email at the maildrop machine RVL4.ecn.purdue.edu at Purdue. (This figure is from Lecture 31 of “Lecture Notes on Computer and Network Security” by Avi Kak)*

Sendmail. Other MTAs include **MMDF**, **Postfix**, **Smail**, **Qmail**, **Zmailer**, **Exchange**, etc.

- An MTA may use either a Mail Delivery Agent (**MDA**) to deliver a received email to the recipient's mailbox, or deliver it directly to the recipient's mailbox. [Note that MTA's main job is server-to-server transmission of email. On the other hand, MDA's job — when MDA is used — is to apply any applicable filters to the email before sending the messages to the clients in the local network.] On **Linux/Unix platforms**, the most commonly used MDA is **Procmail**. Another MDA one hears about is called **Deliver**.
- In typical Linux/Unix environments, the mailbox assigned to a user is the file **/var/mail/user_account** that, although NOT in the home directory of the user, can only be read by the user who owns that mailbox.
- As mentioned earlier, MDA's main job is to apply appropriate filters to the email before it is deposited in the mailboxes of the user accounts. These filters may be at the system level, in which case they can affect all users, or at the level of individual users.
- **On Linux/Unix machines, the filters used by MDA take the form of recipes that are placed in a file named .procmailrc.** The **.procmailrc** file for the filters that are specific to individual users must reside at the top level of the user-account home directories.

- After the email is deposited in a user mailbox as mentioned above, it may be read by the user with the help of an **MUA** (Mail User Agent). Widely used examples of MUAs are Thunderbird, MH, Pine, Elm, Mutt, Outlook, Eudora, Evolution, etc. Informally speaking, an MUA is also frequently referred to as an **an email client**.
- Getting back to how I read my email as shown in Figure 1, I usually execute the two commands

```
ssh kak@rvl4.ecn.purdue.edu  
tail -f Mail/logfile
```

in one of the terminal windows of whatever computer I happen to be working on. As shown in Figure 1, the local email exchange server sends my email to the machine `rvl4.ecn.purdue.edu`. The ‘`tail -f`’ command shows me on a running basis the latest entries created by Procmail in the logfile ‘`Mail/logfile`’. That way, when I so wish, I can see at a glance the decisions being made by my spam filter with regard to the incoming email. The **logfile** that you see mentioned in the second command shown above is created by my Procmail spam filter.

- The rest of this section is for folks who wish to use the Thunderbird MUA on their Ubuntu laptop (or other mobile devices based on Ubuntu) to pick up email from a designated maildrop machine (and to also deliver the outgoing email

emanating from your laptop to the SMTP server running on the maildrop machine or elsewhere in the internet). The material that follows is particularly applicable if you want your spam filter to do its job in the maildrop machine itself. That is, you want the incoming email to be filtered *before* it is made available for pickup at the maildrop machine by an IMAP server. So here we go:

- My maildrop machine happens to be `RVL4.ecn.purdue.edu` and I want the spam filter to be applied at the maildrop machine before it is made available by an IMAP server for pickup by my laptop (or other mobile devices).
- Ordinarily (*this is the mode used by a vast majority of folks*), when an MUA client (like the Thunderbird client) in your laptop picks up email from a maildrop machine, it interacts directly with the IMAP server on the maildrop machine. That creates a very tight coupling between the email client running in your laptop and the mailbox file `/var/mail/user_name` in the maildrop machine where all your email is deposited. As an example of this coupling, when you delete an email in the Thunderbird email client, you can opt for it to also be deleted from the list of messages stored in `/var/mail/user_name` on the maildrop machine. [As previously mentioned, a file such as `/var/mail/user_name` is referred to as the *mailbox*.]
- For reasons having to do with the management of a very large amount of email (including spam) that I receive every day, I did not want the above mentioned coupling between my maildrop machine (`RVL4.ecn.purdue.edu`) and the Thunderbird email client on my laptop. What that implied was that I needed to run Thunderbird off

the laptops's `/var/mail/user_name` as opposed to RVL4's `/var/mail/user_name`.

- This required running the `fetchmail` and `sendmail` programs on the Ubuntu laptop. It is the job of `fetchmail` to serve as a client to the IMAP server on RVL4 — it picks up the new email once every minute from `/var/mail/user_name` on RVL4 and offers it on port 25 of the Ubuntu laptop. Subsequently, `sendmail`, which is constantly looking for input on port 25, picks up the messages offered by `fetchmail` and deposits them in the laptops's mailbox `/var/mail/usr_name`.
- I did not have to change anything in the `sendmail`'s very large config files for the above mentioned behavior by `sendmail`.
- The remaining issue is to get Thunderbird (TB) to work off the mailbox `/var/mail/user_name` in the laptop itself. [To get the TB email client to work directly off an IMAP server on a remote maildrop machine is easy. All you have to do is to enter the IMAP server information and your email address in the remote machine directly in the initial welcome screen you see when you bring up TB in the laptop. But, for reasons already explained, that's not what I wanted.] To get TB to work with the local (meaning, on the laptop itself) mailbox `/var/mail/user_name`, you have to work off the Edit menubutton at the top of the TB GUI and select "Account Settings..." from its drop-down menu. After you click on this selection, you click on "Add Other Account". That brings up a popup, in which you click on "Choose Unix Movemail" and hit "next" and so on. This process will also prompt you for the SMTP server for the outgoing email, which in my case happened to be `smtp.ecn.purdue.edu`. [It is choosing "Unix Movemail" that causes the TB client to work off the mailbox `/var/mail/user_name` on the laptop itself.]

- You might ask: What is **Movemail**? [Before I realized what **Movemail** was, the TB would display in the GUI my `kak@purdue.edu` account that I had created as described above, but without the **Inbox**, **Sent**, **Trash**, etc., folders.] As it turns out, for the TB GUI to make available the **Inbox**, **Sent**, **Trash**, etc., folders, you need to have previously installed the Gnu email utilities that are included in the **mailutils** package that you can install through the Synaptic Package Manager. **Movemail** is one of the utilities in this package. The purpose of **Movemail** — more accurately called **movemail** — is to move messages across mailboxes. [By the way, the others utilities in the Gnu **mailutils** package are: **dotlock** to create lock spool files; **frm** to display “From:” header lines; **from** to display “From:” and “Subject” header lines; **maildag** the mail delivery agent; **mail** the standard `/bin/mail` interface for a mail sender and reader; **messages** for counting the number of messages in a mailbox; **movemail** to move messages across mailboxes; **readmsg** to extract selected messages from a mailbox; and **sieve** a mail filtering protocol.]
- One more thing: You will also be asked for the SSL/TLS based authorizations for SMTP in a screen that you’ll see after you provide information about the SMTP server.

[Back to TOC](#)

31.3 STRUCTURE OF AN EMAIL MESSAGE

- An email consists of three parts:

envelope: This part is usually suppressed by an MUA. [Some MUAs provide you with a menu option to see all the headers, including the routing headers.] It consists of the “conversation” that takes place between a sender MTA and a receiver MTA involving recipient authentication, etc. **As you will see later through examples, the envelope begins with the return address that, if necessary, can be used by the mail exchange servers to bounce the email back to its origination point. This return address is commonly referred to as the “Envelope From” address — to distinguish it from the “Header From” that I’ll mention next. The envelope also contains information as to what email software was used to compose the message.**

header: Contains the “From:”, “To:”, “Cc:”, etc., information. **These are the headers that are shown to you by an email client like the Microsoft Outlook or the Linux Thunderbird.** The string that one sees in the “From:” field is referred to as the “Header From” — to distinguish it from the “Envelope From” mentioned above.

The header of an email message ends at the first empty line encountered from the top. What comes after that empty line is the body of the email. [It is important to know where exactly the header of an email ends and where the body begins. That is because the spam filter rules can be based on just the header, or just the body, or both. For a spam filter rule meant for just the header, the pattern matching operations of the rule are applied to just the header portion of the emails.] Note that the header does NOT tell you the route the email took from the sender to the recipient.

body: This is the part that carries the message of the email. It may also contain multimedia objects.

- Here is a printout of an email as displayed on a terminal by an MUA:

```
Date:      Sat, 14 Feb 2014 19:06:56 CST
To:        kak@ecn.purdue.edu
From:      c-donnelly@northwestern.edu
Subject:   Re: hi...
Return-Path: c-donnelly@northwestern.edu
Delivery-Date: Sat Feb 14 20:07:06 2014
Content-Disposition: inline
X-Originating-Ip: 165.124.28.55
Priority: 3 (Normal)
X-Webmail-User: cdo388@localhost
X-Priority: 3 (Normal)
MIME-Version: 1.0
X-Http_host: lulu.it.northwestern.edu
Reply-To: c-donnelly@northwestern.edu
X-Mailer: EMUmail 5.2.7 (UA Mozilla/4.0 (compatible; MSIE 6.0; Windows NT
          5.1; .NET CLR 1.1.4322))
X-Virus-Scanned-ECN: by AMaVIS version 11 (perl 5.8) (http://amavis.org/)
```

..... Body of email

- For the email shown above, here is a printout of what was actually sent by the MTA to the MDA:

```

From c-donnelly@northwestern.edu Sat Feb 14 20:07:06 2014
Received: from fairway.ecn.purdue.edu (fairway.ecn.purdue.edu [128.46.125.96])
    by rvl4.ecn.purdue.edu (8.12.10/8.12.10) with ESMTP id i1F1758Y006551
    (version=TLSv1/SSLv3 cipher=EDH-RSA-DES-CBC3-SHA bits=168 verify=NOT)
    for <kak@rvl4.ecn.purdue.edu>; Sat, 14 Feb 2014 20:07:06 -0500 (EST)
Received: from lulu.it.northwestern.edu (lulu.it.northwestern.edu [129.105.16.54])
    by fairway.ecn.purdue.edu (8.12.10/8.12.10) with ESMTP id i1F172gN003361
    for <kak@ecn.purdue.edu>; Sat, 14 Feb 2014 20:07:02 -0500 (EST)
Received: (from mailnull@localhost)
    by lulu.it.northwestern.edu (8.12.10/8.12.10) id i1F1718S028285
    for <kak@ecn.purdue.edu>; Sat, 14 Feb 2014 19:07:01 -0600 (CST)
Message-Id: <200402150107.i1F1718S028285@lulu.it.northwestern.edu>
Received: from lulu.it.northwestern.edu (localhost [127.0.0.1]) by lulu.it.northwestern.edu
    id xma028114; Sat, 14 Feb 04 19:06:56 -0600
Content-Type: text/plain
Content-Disposition: inline
Content-Transfer-Encoding: binary
X-Originating-Ip: 165.124.28.55
Priority: 3 (Normal)
X-Webmail-User: cdo388@localhost
To: kak@ecn.purdue.edu
X-Priority: 3 (Normal)
MIME-Version: 1.0
X-Http_host: lulu.it.northwestern.edu
From: c-donnelly@northwestern.edu
Subject: Re: hi...
Date: Sat, 14 Feb 2014 19:06:56 -0600
Reply-To: c-donnelly@northwestern.edu
X-Mailer: EMUmail 5.2.7 (UA Mozilla/4.0 (compatible; MSIE 6.0; Windows NT
    5.1; .NET CLR 1.1.4322))
X-Virus-Scanned-ECN: by AMaVIS version 11 (perl 5.8) (http://amavis.org/)

```

..... Body of email

- In what was sent by the MTA to the MDA, the following is abstracted from the conversation that took place between the

different MTA's as the email was traveling through the internet:

```
Received: from fairway.ecn.purdue.edu (fairway.ecn.purdue.edu [128.46.125.96])  
  by rvl4.ecn.purdue.edu (8.12.10/8.12.10) with ESMTP id i1F1758Y006551  
  (version=TLSv1/SSLv3 cipher=EDH-RSA-DES-CBC3-SHA bits=168 verify=NOT)  
  for <kak@rvl4.ecn.purdue.edu>; Sat, 14 Feb 2014 20:07:06 -0500 (EST)
```

```
Received: from lulu.it.northwestern.edu (lulu.it.northwestern.edu [129.105.16.54])  
  by fairway.ecn.purdue.edu (8.12.10/8.12.10) with ESMTP id i1F172gN003361  
  for <kak@ecn.purdue.edu>; Sat, 14 Feb 2014 20:07:02 -0500 (EST)
```

```
Received: (from mailnull@localhost)  
  by lulu.it.northwestern.edu (8.12.10/8.12.10) id i1F1718S028285  
  for <kak@ecn.purdue.edu>; Sat, 14 Feb 2014 19:07:01 -0600 (CST)
```

- Also note the first line of what MTA sends MDA:

```
From c-donnelly@northwestern.edu Sat Feb 14 20:07:06 2014
```

For an email to be recognized as legal by an MTA, its very first line must begin with “From”. There can be no punctuation marks attached to this word. In other words, it can only be followed by a space.

You might ask, what happens if the body of an outgoing message contain the word “From” at the beginning of a line? To make sure that MTAs are not confused by this, an MSA typically prefixes such a “From” with the character “>”. Obviously, such a problem does not arise if you have asked your email client to send messages formatted according to, say, HTML. [As was mentioned earlier, MSA stands for “Mail Submission

Agent”. Your email client sends an outgoing email to an MSA and it is the MSA’s job to then submit it to an MTA possibly through an SMTP server.]

- Also note that the name of the final recipient is present in the conversation that takes place between the MTA’s at the Northwestern end and at Purdue’s `fairway.ecn.purdue.edu` machine. The name of the recipient is also present in the conversation that takes place between Purdue’s `fairway` machine and the local RVL4 machine.
- **It is the recipient’s name in the envelope part of an email that determines where an email ends up and NOT what shows up in the To: header in the header part of an email.**
- So you can see why you can get email even if your name shows up nowhere in any of the headers you can see on your computer. Here is an example of one such spam email I received:

```
From leemenjung@kjbd.net Thu Feb 19 10:19:02 2014
Received: from drydock.ecn.purdue.edu (drydock.ecn.purdue.edu [128.46.112.249])
by rvl4.ecn.purdue.edu (8.12.10/8.12.10) with ESMTP id i1JFJ1j4025944
(version=TLSv1/SSLv3 cipher=EDH-RSA-DES-CBC3-SHA bits=168 verify=NOT)
for <kak@rvl4.ecn.purdue.edu>; Thu, 19 Feb 2014 10:19:02 -0500 (EST)
Received: from 128.46.112.249 ([61.38.114.147])
by drydock.ecn.purdue.edu (8.12.10/8.12.10) with SMTP id i1JFIImFj028889;
Thu, 19 Feb 2014 10:18:49 -0500 (EST)
Received: from [27.22.18.140] by 128.46.112.249 with ESMTP id <229528-89751>; Thu, 19 Feb 2014 17:13:
Message-ID: <joh3yy$x$-$317$2c-v--21n@hzh6.9t>
From: "leemenjung" <leemenjung@kjbd.net>
Reply-To: "leemenjung" <leemenjung@kjbd.net>
```

```

To: jiy@ecn.purdue.edu
Subject:
Date: Thu, 19 Feb 04 17:13:48 GMT
X-Mailer: Microsoft Outlook Express 5.00.2919.6700
MIME-Version: 1.0
Content-Type: multipart/alternative;
boundary="0.D6.._EFOB97BFE__AA._6_"
X-Priority: 3
X-MSMail-Priority: Normal
X-Virus-Scanned-ECN: by AMaVIS version 11 (perl 5.8) (http://amavis.org/)

--0.D6.._EFOB97BFE__AA._6_
Content-Type: text/plain;
Content-Transfer-Encoding: quoted-printable

<html>
  <TABLE cellpadding=3D'0' cellspacing=3D'0' border=3D0 align=3D'center'>=

    <TR>
      <TD height=3D'50' bgcolor=3D'#FFFFFF' align=3D'center' valign=3D=
'middle'>

        <a href=3D"http://nipponbog.com/partner/recom.asp?recome_id=3Dstart"=
target=3D"_blank"><img src=3D"http://nipponbog.com/partner/email/email2=
/1.jpg" border=3D"0"></a>
      </TD>
    </TR>
  </TABLE>
</html>
oada slh vwudbxr sodb frjmh
bs arf
ohf
vjkutctg
yzmyzfujadg
ua
uq ffwd
uh

--0.D6.._EFOB97BFE__AA._6_--

```

In the spam mail shown above, my name shows up only in the envelope part of the headers.

- Going back to the first **c-donnelly** email I showed you in this section, if I examined what the MUA actually stored for that

message (as opposed to what it displayed in the GUI), it would be something like

```
Return-Path: c-donnelly@northwestern.edu
Delivery-Date: Sat Feb 14 20:07:06 2014
Received: from fairway.ecn.purdue.edu (fairway.ecn.purdue.edu [128.46.125.96])
    by rvl4.ecn.purdue.edu (8.12.10/8.12.10) with ESMTP id i1F1758Y006551
    (version=TLSv1/SSLv3 cipher=EDH-RSA-DES-CBC3-SHA bits=168 verify=NOT)
    for <kak@rvl4.ecn.purdue.edu>; Sat, 14 Feb 2014 20:07:06 -0500 (EST)
Received: from lulu.it.northwestern.edu (lulu.it.northwestern.edu [129.105.16.54])
    by fairway.ecn.purdue.edu (8.12.10/8.12.10) with ESMTP id i1F172gN003361
    for <kak@ecn.purdue.edu>; Sat, 14 Feb 2014 20:07:02 -0500 (EST)
Received: (from mailnull@localhost)
    by lulu.it.northwestern.edu (8.12.10/8.12.10) id i1F1718S028285
    for <kak@ecn.purdue.edu>; Sat, 14 Feb 2014 19:07:01 -0600 (CST)
Message-Id: <200402150107.i1F1718S028285@lulu.it.northwestern.edu>
Received: from lulu.it.northwestern.edu (localhost [127.0.0.1]) by lulu.it.northwestern.edu
    id xma028114; Sat, 14 Feb 04 19:06:56 -0600
Content-Type: text/plain
Content-Disposition: inline
Content-Transfer-Encoding: binary
X-Originating-Ip: 165.124.28.55
Priority: 3 (Normal)
X-Webmail-User: cdo388@localhost
To: kak@ecn.purdue.edu
X-Priority: 3 (Normal)
MIME-Version: 1.0
X-Http_host: lulu.it.northwestern.edu
From: c-donnelly@northwestern.edu
Subject: Re: hi...
Date: Sat, 14 Feb 2014 19:06:56 -0600
Reply-To: c-donnelly@northwestern.edu
X-Mailer: EMUmail 5.2.7 (UA Mozilla/4.0 (compatible; MSIE 6.0; Windows NT
    5.1; .NET CLR 1.1.4322))
X-Virus-Scanned-ECN: by AMaVIS version 11 (perl 5.8) (http://amavis.org/)
```

..... Body of email

- With regard to the printout shown above, recall I said earlier

that for an email to be legal, its first line must start with “From”, which in turn must be followed by a blank space. The printout is meant to convey to you the fact that an MUA may modify the very first “From” line into two separate lines, one for “Return-Path” and the other for “Delivery-Date”.

- So what an MTA sends an MDA may not be the same as what the MUA stores for the email and that, in turn, may not be the same as what the MUA actually shows you on the screen.

[Back to TOC](#)

31.4 HOW SPAMMERS ALTER THE EMAIL HEADERS — A CASE STUDY

- I will now present an instance of a spam email in which the main **From** header at the top of the email record was faked. Note that the receiving MDA has converted the keyword **From** into the **Return-Path** header label.
- Shown below is an email that was received by my Purdue account on April 4, 2010:

```
Return-Path: cossacksrg1@ralvm29.vnet.ibm.com
Delivery-Date: Sun Apr 4 12:36:10 2010
Received: from mx03.ecn.purdue.edu (mx03.ecn.purdue.edu [128.46.105.218])
by rvl4.ecn.purdue.edu (8.14.4/8.14.4) with ESMTP id o34GaAhE013679
(version=TLSv1/SSLv3 cipher=DHE-RSA-AES256-SHA bits=256 verify=NOT)
for <kak@rvl4.ecn.purdue.edu>; Sun, 4 Apr 2010 12:36:10 -0400 (EDT)
Received: from 114-24-88-69.dynamic.hinet.net (114-24-88-69.dynamic.hinet.net [114.24.88.69])
by mx03.ecn.purdue.edu (8.14.4/8.14.4) with ESMTP id o34GZ2k8020095;
Sun, 4 Apr 2010 12:35:23 -0400
Received: from 114.24.88.69 by e33.co.us.ibm.com; Mon, 5 Apr 2010 00:34:59 +0800
Message-ID: <000d01cad414$c4404060$6400a8c0@cossacksrg1>
From: "Minerva Souza" <cossacksrg1@ralvm29.vnet.ibm.com>
To: <eatabay@ecn.purdue.edu>
Subject: ecn.purdue.edu account notification
Date: Mon, 5 Apr 2010 00:34:59 +0800
MIME-Version: 1.0
Content-Type: multipart/mixed;
    boundary="-----_NextPart_000_0006_01CAD414.C4404060"
X-Priority: 3
X-MSMail-Priority: Normal
X-Mailer: Microsoft Outlook Express 6.00.2900.2180
X-MimeOLE: Produced By Microsoft MimeOLE V6.00.2900.2180
X-ECN-MailServer-VirusScanned: by amavisd-new
X-ECN-MailServer-Origination: 114-24-88-69.dynamic.hinet.net [114.24.88.69]
X-ECN-MailServer-SpamScanAdvice: DoScan
Status: RO
```

X-Status:
X-Keywords:
X-UID: 7

This is a multi-part message in MIME format.

```
-----=_NextPart_000_0006_01CAD414.C4404060
Content-Type: text/plain;
format=flowed;
charset="iso-8859-1";
reply-type=original
Content-Transfer-Encoding: 7bit
```

Dear Customer,

This e-mail was send by ecn.purdue.edu to notify you that we have temporanly prevented access to your account.

We have reasons to beleive that your account may have been accessed by someone else. Please run attached file a

(C) ecn.purdue.edu

```
-----=_NextPart_000_0006_01CAD414.C4404060
Content-Type: application/zip;
name="Instructions.zip"
Content-Transfer-Encoding: base64
Content-Disposition: attachment;
filename="Instructions.zip"
```

```
UESDBBQAAGaIAFkQhDwZeJaCR18AADVzAAAQAAASW5zdHJ1Y3Rpb25zLmV4Ze38BVQfTbcnjP5x
C04ElwDBHUJwtxDc3d3d3dXQNBA8EhENZd3R0S/DZPnv0e98ic03dm7pr5vjW1dknvqv5tqapd
3fInIaOeC4IagUCQQH55AYGaQX8SP+j/e3odi0TUggSghxshaQb7NEKiaGrmQGxrb2Nir2dFbKBn
bW3jSKxvRGzvZE1sZk0sLKNAbGVjaESPiPjmHeh/LsmKgECfwKBAYFiNUv/CWwchg8GDQSH8ZRDk
30yIvzP031aBgf7KkH93/0sNcvx7HJDA/ypR/sZA+QcWyj/JJwbwuF8bsCCQLiLof10CcIn/i256
RyPXV1WNwf/JNoh/Owa4X5fe3lDPUQ8Euv0b8y+7of/tOMAb/PR/hv2xBebvcTD/YVwnvb2DvQHo
.....
.....
.....
```

```
-----=_NextPart_000_0006_01CAD414.C4404060--
```

- If you examine the headers, you will see that the email was generated by 114.24.88.69. If you enter this address in <http://www.ip2location.com> window, you will see that this address belongs to “Chunghwa Telecom Data Communication Business Group” in Taipei, Taiwan. Obviously, it is not easy for me to tell whether this domain is hosting an anonymizing email server

that is acting as a mail forwarder for third-party folks, or being more directly complicit in sending out the spam.

- You will also notice in the email message shown above that it contains a fake “**Received: from**” line that seems to indicate that the email was received by a server named **e33.co.us.ibm.com** from the address 114.24.88.69 in Taiwan. This line is fake because higher up in the email header you can see that the mail exchange server for the **ecn.purdue.edu** domain received the email directly from 114.24.88.69.
- My email log file indicated that this email slipped through my powerful spam filter, meaning that it fell off the bottom of my **.procmailrc** file. That is because the main text portion of the message in this email does not contain anything offensive. [I could easily include another recipe in my spam filter that would delete a message that contained a zip attachment consisting of just **.exe** executables. But then I would not have found this gem.]
- When I unzipped the attachment in the email shown above, it contained only a single file called **Instructions.exe**. Executing the command “**file Instructions.exe**” yielded the following answer:

```
PE32 executable for MS Windows (GUI) Intel 80386 32-bit
```

indicating that the executable was meant for a Windows machine. About the MS DOS PE header shown above, the

Windows NT OS introduced a new executable file format called the Portable Executable (PE) file format. It retains the old familiar MZ header from MS-DOS, as you will see in the partial hexdump of the file presented below.

- Another way to confirm the fact that this file is a Windows executable is by looking at its hexdump:

```
/usr/bin/hexdump -C Instructions.exe | more
```

As shown below, in the very first line you can see the telltale “MZ” marker that is the beginning of a MS-DOS PE header.

```
00000000  4d 5a 90 00 03 00 00 00  04 00 00 00 ff ff 00 00  |MZ.....|
00000010  b8 00 00 00 00 00 00 00  40 00 00 00 00 00 00 00  |.....@.....|
00000020  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |.....|
00000030  00 00 00 00 00 00 00 00  00 00 00 00 b8 00 00 00  |.....|
00000040  0e 1f ba 0e 00 b4 09 cd  21 b8 01 4c cd 21 54 68  |.....!..L.!Th|
00000050  69 73 20 70 72 6f 67 72  61 6d 20 63 61 6e 6e 6f  |is program canno|
00000060  74 20 62 65 20 72 75 6e  20 69 6e 20 44 4f 53 20  |t be run in DOS |
00000070  6d 6f 64 65 2e 0d 0d 0a  24 00 00 00 00 00 00 00  |mode....$.....|
00000080  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |.....|
*
.....
.....
```

- When I uploaded the malicious file to the online virus analysis tool at <http://www.virustotal.com>, I received a report that it was a well-known virus. The report also included the virus signature and other attributes of the virus.

[Back to TOC](#)

31.5 SPF, DKIM, and DMARC — The Three Important Email Authentication Protocols

- Probably the main reason that bad guys are so fond of using email to spread malware is because it has been so easy to fake the headers in the messages. The sender could pretend to be anyone simply by changing the entry in the “From:” header of the message. **As you know already, the name and the email address in the “From:” header do not play any role in how a message is routed.**
- And, with only a bit more difficulty, the sender would also be able to alter the “From:” header in the envelope — although this would require the cooperation of the Internet Service Provider where email is originating. As you saw in Lecture 16, it is difficult to spoof your source IP address with ISPs that have implemented BCP 38. **Obviously, such things would not matter when state actors are involved or when the bad guys are routing their email through countries with lax laws.**
- Because of the different roles played by what goes into the “From:” header in an email and the same information that is inserted in the envelope, people have wondered for a long time if

one way to clamp down on spam would be for the on-line mail forwarding agents to at least verify that the email origination domain mentioned in the envelope was the same as the domain mentioned in the “From:” header. As implied by the previous bullet, the main problem with just checking that the two domain names are the same is that both the “From” addresses can be faked by cyber criminals.

- Fortunately, we now have the three email authentication protocols that are capable of doing a good job of catching fake email. While these protocols have cut down on spam, it is still out there in great abundance — as I will demonstrate in the next section in this lecture.

SPF (Sender Policy Framework) This protocol allows an organization to specify who is allowed to send email on behalf of the domain to which the organization belongs. This is done by placing appropriate entries in the DNS servers as I’ll explain in the rest of this section.

DKIM (Domain Keys Identified Mail) In an email exchange between two servers, it allows the receiving server to carry out cryptographic authentication of the fact that the email was actually sent by the sending server. The sending server notifies the receiving server as to which parts of the email it has hashed and then encrypted with its private key. Subsequently, the receiving receiver can authenticate the sender by verifying the signature with the sender’s public

key. Obviously, the part of the email that is digitally signed at the sending end must remain invariant during transit. A simple entry for, say, the “Header From” would work for that purpose.

DMARC (Domain-based Message Authentication Reporting and Conformance)

The main purpose of this protocol is to verify that the domain in the “Envelope From” is the same as the domain in the “Header From”. **This is referred to as “domain alignment”**. Information gleaned from the DNS servers plays an important role in the verification of domain alignment. DMARC can carry out domain alignment check both on the basis of SPF information gleaned from DNS and on the basis of DKIM signatures.

- For their functioning, these protocols depend on the TXT records in DNS servers. In Lecture 17 you have already seen Resource Records that are returned by a DNS server in response to a name or pointer query. The TXT records are in addition to those. **In general, a TXT record stored in a DNS server is any information at all that a domain administrator wants to make available regarding their domain.** This flexibility about the TXT records has allowed them to be put to good use for email authentication.
- In the rest of this section, I’ll provide additional details about the three email authentication protocols mentioned above.,

SPF:

- Of the three protocols, SPF is particularly important in the sense that the other two protocols, DKIM and DMARC, would not be able to function without SPF.
- For an organization that wants to use SPF in its outgoing email (so that the others can make sure that the email that claims to be coming from the said organization is telling the truth), the very first thing it has to do create an “SPF TXT” record in the DNS that is the authoritative name server for the domain of the organization.
- Let’s say you are running a small business and you have registered your domain with a domain-name registrar. The SPF record you will need for the DNS entry for your domain name will look like:

```
v=spf1 ip4:xxx.xxx.xxx.xxx ip4:xxx.xxx.xxx.xxx include:my_business_domain.com -all
```

in which you can list any number of IP addresses that your business would be using for email communications. You can also express your IP range by using the CIDR notation:

`xxx.xxx.xxx.xxx/yy` where **yy** would be for the number of bits from the left that remain the same for the IP addresses your business is using. Ordinarily, they would be the same as the mask for the intranet you are using. The acronym CIDR stands for “Classless Inter-Domain Routing”.

- You can check the SPF record for any domain by using the `dig` and `nslookup` commands for querying a DNS, but you have to ask for the TXT record. For example, the command “`dig purdue.edu txt`” returns the following value for the SPF record for the “purdue.edu” domain:

```
v=spf1 a:smtp.purdue.edu ip4:128.210.210.98 ip4:128.210.210.99 ip4:128.210.40.161 ip4:128.210.40.162
include:a._spf.brightspace.com include:spf.protection.outlook.com include:ne16.com
include:_spf.qualtrics.com include:_tuf-spf.touchnet.com include:_spf.salesforce.com ?all"
```

- All of the information shown above is in a single line. I believe that in the TXT record that is entered in the DNS database, if the line you wish to enter is too long, you are allowed to have a sequence of quoted strings that the system joins together for the value of the record.
- Through the record shown above, Purdue is declaring that any email emanating from the domain “purdue.edu” will come from the IPv4 addresses listed in the record.
- Pay attention to the “include” entries in the SPF record shown above. These entries play an important role in a practical implementation of email authentication with the SPF protocol. These entries mean that when a receiving mail exchange server is trying to verify the origin of an email, it must also query the SPF records for the domain names listed as the “include” entries. To elaborate, as a Purdue student, when you send an

email to someone outside the Purdue domain, that email actually goes through the Microsoft Outlook email servers — because Purdue pays Microsoft to handle all its email. So even though the recipient of your email is going to see your Purdue email address in the “From:” header in their email client, the “Envelope From” will carry an IP address that has been assigned to the “outlook.com” domain. So any attempt at verification of the origin of your email would need to reconcile the “Envelope From” IP address with the SPF record for the “purdue.edu” domain.

- As shown for the “purdue.edu” domain, an SPF record MUST begin with the string “v=spf1.”. As you will see, in general, a DNS server will include several such records in the overall TXT record. Some of those are needed for the functioning of the DKIM and DMARC protocols.
- Let’s now focus on the entry “?all” in the SPF record for the “purdue.edu” domain. You are allowed three possibilities here:
 - all : means that if the source IP address in the email received from the sending MTA is not one of those allowed by the SPF record, consider that email to have failed the verification test. This is considered to be a **hard fail**.
 - ~all : This is a **soft fail** version of the above. In this case, the receiving MTA may tag the email as Spam.
 - ?all : This is a **neutral policy** version of the previous two entries, implying that the receiving MTA can dispose of the email any way it wants to.

- For a more complex SPF record, let's look at the output of the command "**dig cnn.com txt**". It yields the following information:

```
v=spf1 include:cnn.com._nspf.vali.email include:%{i}._ip.%{h}._ehlo.%{d}._spf.vali.email \
include:mail.zendesk.com ~all"
```

- You might ask: What are these %i, %h, and %d entries in the record? These are referred to as the SPF Macros that make it easier to create **dynamic SPF policies**. Basically, they allow for variables to be included within a policy. For the meanings of these three macros:

%s : The sender of the received email. For example, "bob@purdue.edu"

%i : The source IP address

%d : The authoritative domain of the SPF policy being processed

- Listed above are just three of the macros. The following website presents a more complete listing:

<https://www.jamieweb.net/blog/using-spf-macros-to-solve-the-operational-challenges-of-spf/>

This is a good time to also mention the document RFC 7208 that is the official standard for the SPF protocol:

<https://www.rfc-editor.org/rfc/rfc7208>

- What should an email service provider do when it receives an email that does not pass the checks mentioned in this section? That depends on the service provider, as you would expect. A large volume email service provider like googlemail is likely to reject it. [I know if from personal experience as I'll explain in class.] On the other hand, a smaller email service provider, such as the one provided by the “purdue.edu” domain, **may just quarantine the email and let the recipient know.**

DKIM:

- If you make dig queries on the TXT records for a random assortment of the domains, you are likely to see the SPF record for practically all the domains, but not as commonly for the DKIM or DMARC records. While all the mega domains — from Google, Microsoft, Yahoo, etc. — have implemented all three protocols, to the best of what I know a majority of the smaller domains have not done so. It could be because DKIM is more complex and, for a sys-admin to be comfortable with it, they have to be comfortable with the mechanics of public-key crypto. **Many otherwise wonderful folks who specialize in systems side of computer security are not always at ease with the ins and outs of cryptographic algorithms.**
- The basic purpose of a DKIM TXT record is to store the public key that a sending MTA has used for digitally signing the hash

of a designated field in the email being processed by the receiving MTA.

- The syntax of a DKIM TXT entry in a DNS server looks like:

```
v=DKIM1; p=yourPublicKey
```

where “yourPublicKey” is the Base64 encoded public-key. A domain is allowed to store multiple public keys in the TXT record associated with the domain, with each public key identified by a **selector** string. This selector/domain combination would be included in the DKIM header that the sending MTA would insert in an outgoing email. More on selectors after I have shown you an example of a DKIM header in an email message received by Purdue MTA.

- First, let’s address the following question: How does an email carry along with it the DKIM signature that can be used by the receiving MTA for its authentication? Obviously, as you would expect, that is accomplished by inserting additional headers in the email. To illustrate these headers, I will use as an example a spam email I received sometime back:

```
DKIM-Signature: v=1; a=rsa-sha256; q=dns/txt; c=relaxed/simple;  
s=dvogjbbaa3ou3tduyzvvyu4rj5tkuzdi4h;  
d=amazonses.com;  
t=1680192085;  
h=From:To:Subject:Date:Message-ID:MIME-Version:Content-Type:Feedback-ID;  
bh=rk+ydIpW2IkheYVl5mlWqJq+pmZHSw1OHCOWxoRMtKA=;
```

```
b=aFWk/jM8MrDMW4jinnQa+HkkTG5XGnN5Wu35aQ+KLn5kOPHwhLe1Bnj0dQW1etAK
DEPREKHvnxVfrAlyjqvPtZqxoASBa70kluS6x5uy/nyn01SmvfTBT7iiR1IFrE01A4d
ixMMJ97b2dv8mXaoa9s6bH+I1YQWLq6wp3RUUboU=
```

where:

v= shows which version of DKIM is in use.

d= is the domain name of the sender.

s= is the selector that the receiving server should use to look up the DNS record

h= lists the header fields that are used to create the digital signature

bh= is the hash of the email body.

a= is the algorithm used to compute the digital signature

b= is the digital signature, generated from h and bh and signed with the private

- Note that the receiving MTA can authenticate the email even before the body of the email has been loaded — since the signature is based on just the headers and since DKIM header includes the signature.
- Here is the value of the selector field in the example shown above “dvogjbba3ou3tduyzvyyu4rj5tkuzdi4h”. Using this selector value, I can query the domain mentioned in the DKIM signature shown above by making the following dig query:

```
dig -t TXT +noall +answer dvogjbba3ou3tduyzvyyu4rj5tkuzdi4h._domainkey.amazonses.com
```

Note that I had to insert the string “**._domainkey**” between the selector and the domain name. The above query elicits the following answer:

```
dvogjbba3ou3tduyzvyu4rj5tkuzdi4h._domainkey.amazonses.com. 300 IN CNAME \
                                                                    dvogjbba3ou3tduyzvyu4rj5tkuzdi4h.dkim.amazonses.com.
dvogjbba3ou3tduyzvyu4rj5tkuzdi4h.dkim.amazonses.com. 3599 IN TXT
"p=MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCmEG+v3Zj98RvPBQziW0q+gzy/+DHZRE81/08XXnUvHTvQxV23jRV40psn\
Qkeby+mLwISw4drbLUElKELaUXqVkVk4GEk5XFB5TujADW8IR0nX7mb0d+ZVQ1RpX6K7LdN6nlxnFFIBSuvI\
                                                                    yFOIX8xRwtzj6xD1DKZxCP+qFX/cQIDAQAB"
```

The value of the “p=” field is the public key associated with that selector. The receiving MTA can use this public key to authenticate the actual signature shown as the value of the “b=” field in the email DKIM header.

- I find it interesting that whereas I am able to check for a selected public key at the **amazonses.com** domain, if I just ran the query **dig amazonses.com txt**, the record that is returned does not include any DKIM entries.

DMARC:

- The DMARC protocol sits on top of the SPF and DKIM protocols. As mentioned earlier, at this moment in 2024, the adoption of SPF appears to be more widespread than that of DKIM. So, you are not likely to see DMARC email headers any more frequently than the DKIM headers.

- The main purpose of the DMARC protocol is to state a policy as to what the receiving MTA should do to an email should their SPF and DKIM based verifications fail. The DMARC policy will state whether such an email should be rejected, or be declared as spam while being forwarded to its recipient, or should the verification failure be just ignored.
- To see an example of a DMARC TXT record, here is what's elicited by the command `dig _dmarc.cisco.com txt`:

```
v=DMARC1; p=quarantine; pct=0; fo=1; ri=3600;  
rua=mailto:ynldvgsr@ag.dmarcian.com; ruf=mailto:ynldvgsr@fr.dmarcian.com;
```

where the value of the “p” for “policy” field indicates what should be done by the receiving MTA should there be a failure of the SPF and/or DKIM verification. The tag “ruf” stands for “Report URI on Failure”. The one or more email addresses associated with this tag will receive a message should there be verification failure — the sender MTA would like to know just in case any errors in the SPF or the DKIM entries are causing the failure. The “rua” stands for “Report URI for Aggregate”. Both these tags are optional but very important to make sure that DMARC is working correctly.

[Back to TOC](#)

31.6 Spam Despite SPF, DKIM, and DMARC — A Case Study

- Despite the three great email authentication protocols, there is still a lot of spam out there. What follows is an example I received March 29, 2023.
- While the email looks legit from the standpoint of its lending to verification by email authentication protocols, **it is still spam in the worst sense of what's meant by that word.**
- **It is spam because its message is a big lie.** I will show screenshots of the content of the message after the headers presented below.
- The headers shown below includes the DKIM signature, implying that the email is not lying about where it came from. Nonetheless, it is trying to deceive its recipient.

Return-Path: <noreply@serverhost.com>

Received: from mx04.ecn.purdue.edu (mx04.ecn.purdue.edu [128.46.154.93])

by rvl4.ecn.purdue.edu (8.14.7/8.14.7) with ESMTTP id 32TDedrA035957

(version=TLSv1/SSLv3 cipher=ECDHE-RSA-AES256-GCM-SHA384 bits=256 verify=NOT)

for <kak@rvl4.ecn.purdue.edu>; Wed, 29 Mar 2023 09:40:39 -0400

Received: from 6217806.dustlessblastingmarketingservices.com (6217806.dustlessblastingmarketingservices.com [16

by mx04.ecn.purdue.edu (8.15.2/8.15.2/Debian-22ubuntu3) with ESMTTP id 32TDeIRk1242208

```
(version=TLSv1.2 cipher=ECDHE-RSA-AES256-GCM-SHA384 bits=256 verify=NOT)
for <kak@ecn.purdue.edu>; Wed, 29 Mar 2023 09:40:39 -0400
DKIM-Signature: v=1; a=rsa-sha256; q=dns/txt; c=relaxed/relaxed;
d=minutemenmb.com; s=default; h=Content-Type:MIME-Version:Message-ID:Date:
Subject:To:From:Sender:Reply-To:Cc:Content-Transfer-Encoding:Content-ID:
Content-Description:Resent-Date:Resent-From:Resent-Sender:Resent-To:Resent-Cc:
:Resent-Message-ID:In-Reply-To:References:List-Id:List-Help:List-Unsubscribe:
List-Subscribe:List-Post:List-Owner:List-Archive;
bh=tBHK8ChTl7AtB07N5+SoPb5/fg35myvIWEcyovONb9Q=; b=fySZyVgkW2ejJVTQlQmI7HK3QL
xQeIJff/o1/5Sx0XY9+U7p0DF6uHxM8mYMHYED1ZHdCUePM9M2wqfXLPu5qkaG+XwA8DGdzaBMZJ
U6ZG/rQysI3rSrWTF8ZDHLqsI4EbE4D0wQWZ3YxyNvKzD0f35AxSNcqUs75vbr8ReG5naGuIbEpRs
9isaTbD+VWS9R3uor9a90nButhqI1FNZrKEx4EfN9uPSJj3+U3RIicqiuRaVF5BnWeBrQp1rjHNvN
LeKBuHvGSt3R0UQ3P6D4qE9lJ9KeyXvHXo03M1u30SqtqrZMX1EjvjiaJ/z36PKY0ti9rBQ8GhfAW
10JXLlsw==;
```

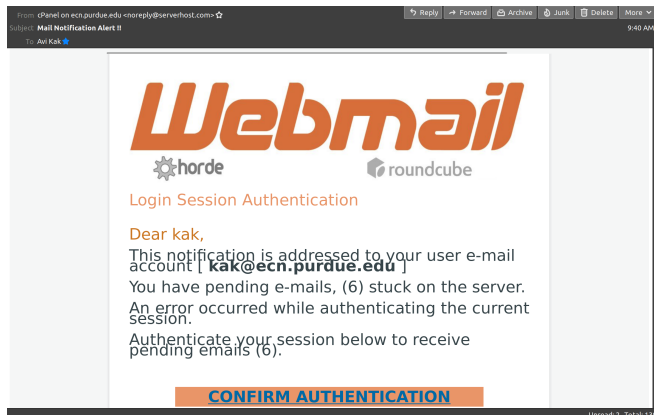
....
....

```
From: "cPanel on ecn.purdue.edu" <noreply@serverhost.com>
To: kak@ecn.purdue.edu
Subject: Mail Notification Alert !!
Date: 29 Mar 2023 13:40:16 +0000
Message-ID: <20230329134016.2E6D3C9A0B761D6F@serverhost.com>
MIME-Version: 1.0
Content-Type: multipart/related;
boundary="-----_NextPart_000_0012_03DA8748.35CEDE87"
```

...
...
...
...
...
...

```
AACxjwv8YQUAAAAJcEhZcwAADsMAAA7DAcdvqGQAAP+1SURBVHhe7P2HfxxX9T8OP//G83s9
v+8HiG0rrdpKq2a5xI4TpxESS0AkJLG1bbMzuyvJJU4FagRCCwQCgRBq6L18IJAEQkK6e5NV
```

- I have shown the message in the email through the screenshots on the next page. The message starts out by saying that I have pending six emails on the email server — a big lie. Then it wants me to click on a GUI button to retrieve the messages.



This email has been sent to kak@ecn.purdue.edu because it contains important information about your account. If you previously unsubscribed from Identity Guard® marketing emails, you will no longer receive special offers, but will continue to receive emails related to your account. If you believe you received this email in error, please send it to our customer service team at ecn.purdue.edu

We will never ask you for personal information in an e-mail. We respect your privacy. If you no longer wish to receive Identity Guard® marketing emails, you can unsubscribe at any time.

© 2023 ecn.purdue.edu Inc.

Click [here](#) to unsubscribe!

- What follows is another example of a “legit” spam. It is targeted for me by my name. Obviously, they purchased a more expensive email list (as opposed to thousands-of-addresses-for-a-dollar lists created by scraping the internet) that shows full names and their email addresses.
- What makes this sample interesting is that it shows how it passed the SPF check by Purdue ITaP.

```
Return-Path: <0109018733409dd0-1e7624b7-31c6-42c0-aec9-3a4722ab3b3c-000000@ap-south-1.amazonses.com>
Received: from mx01.ecn.purdue.edu (mx01.ecn.purdue.edu [128.46.154.90])
by rvl4.ecn.purdue.edu (8.14.7/8.14.7) with ESMTTP id 32UG1Q03019898
(version=TLSv1/SSLv3 cipher=ECDHE-RSA-AES256-GCM-SHA384 bits=256 verify=NOT)
for <kak@rvl4.ecn.purdue.edu>; Thu, 30 Mar 2023 12:01:26 -0400
Received: from xppmailspam11.itap.purdue.edu (xppmailspam11.itap.purdue.edu [128.210.1.215])
by mx01.ecn.purdue.edu (8.15.2/8.15.2/Debian-22ubuntu3) with ESMTTP id 32UG1QFa1362014
(version=TLSv1.2 cipher=ECDHE-RSA-AES256-GCM-SHA384 bits=256 verify=NOT)
for <kak@ecn.purdue.edu>; Thu, 30 Mar 2023 12:01:26 -0400
Authentication-Results: xppmailspam11.itap.purdue.edu; spf=Pass smtp.mailfrom=0109018733409dd0-1e7624b7-31c6-42c0-aec9-3a4722ab3b3c-000000@ap-south-1.amazonses.com
Received-SPF: Pass (xppmailspam11.itap.purdue.edu: domain of
0109018733409dd0-1e7624b7-31c6-42c0-aec9-3a4722ab3b3c-000000@ap-south-1.amazonses.com
designates 76.223.180.1 as permitted sender)
identity=mailfrom; client-ip=76.223.180.1;
receiver=xppmailspam11.itap.purdue.edu;
envelope-from="0109018733409dd0-1e7624b7-31c6-42c0-aec9-3a4722ab3b3c-000000@ap-south-1.amazonses.com";
```



```

x-sender="0109018733409dd0-1e7624b7-31c6-42c0-aec9-3a4722ab3b3c-000000@ap-south-1.amazonaws.com";
x-conformance=spf_only; x-record-type="v=spf1";
x-record-text="v=spf1 ip4:199.255.192.0/22
ip4:199.127.232.0/22 ip4:54.240.0.0/18 ip4:69.169.224.0/20
ip4:23.249.208.0/20 ip4:23.251.224.0/19 ip4:76.223.176.0/20
ip4:54.240.64.0/19 ip4:54.240.96.0/19 ip4:52.82.172.0/22
ip4:76.223.128.0/19 -all"
Received-SPF: Pass (xppmailspam11.itap.purdue.edu: domain of
postmaster@c180-1.smtp-out.ap-south-1.amazonaws.com
designates 76.223.180.1 as permitted sender) identity=helo;
client-ip=76.223.180.1;
receiver=xppmailspam11.itap.purdue.edu;
envelope-from="0109018733409dd0-1e7624b7-31c6-42c0-aec9-3a4722ab3b3c-000000@ap-south-1.amazonaws.com";
x-sender="postmaster@c180-1.smtp-out.ap-south-1.amazonaws.com";
x-conformance=spf_only; x-record-type="v=spf1";
x-record-text="v=spf1 ip4:76.223.180.1 -all"
...
...
..
Received: from c180-1.smtp-out.ap-south-1.amazonaws.com ([76.223.180.1])
by xppmailspam11.itap.purdue.edu with ESMTP/TLS/ECDHE-RSA-AES128-GCM-SHA256; 30 Mar 2023 12:01:25 -0400
DKIM-Signature: v=1; a=rsa-sha256; q=dns/txt; c=relaxed/simple;
s=dvogjbaa3ou3tduyzyu4rj5tkuzdi4h; d=amazonaws.com; t=1680192085;
h=From:To:Subject:Date:Message-ID:MIME-Version:Content-Type:Feedback-ID;
bh=rk+ydIpW2IkheYVl5mlWqJq+pmZHSw10HCOWxoRMtKA=;
b=aFWk/jM8MrDMW4jinnQa+HkkTG5XGnN5Wu35aQ+KLn5kOPHwhLe1BnjOdQW1etAK
DEPRReKHvnxVfrAlyjqvPtZqxoASBa70kluS6x5uy/nyn01SmvftBT7iiR1IFrE01A4d
ixMMJ97b2dv8mXaoa9s6bH+I1YQWLq6wp3RUUboU=
Return-Receipt-To: "Natalie Horne" <nataliehorne@xploreconferences.com>
From: "Natalie Horne" <nataliehorne@xploreconferences.com>
To: <kak@purdue.edu>
Subject: Keynote Speaker invitation for Nanotechnology conference 2024 - Dubai, UAE
Date: Thu, 30 Mar 2023 16:01:25 +0000
Message-ID: <0109018733409dd0-1e7624b7-31c6-42c0-aec9-3a4722ab3b3c-000000@ap-south-1.amazonaws.com>
Content-Language: en-in
...
...

```

This is a multipart message in MIME format.

```

-----=_NextPart_000_8EAD_01D9634E.FB508F60
Content-Type: text/plain;
charset="us-ascii"
Content-Transfer-Encoding: 7bit

```

Dear Dr. Avinash Kak,

I hope this message finds you well and in good health. On behalf of the organizing committee, it is my pleasure to invite you to the International Conference on Nanotechnology and Material Science (ICNMS - 2024), which will be held in Dubai, UAE from March 20 to 22, 2024.

...

[Back to TOC](#)

31.7 A VERY BRIEF INTRODUCTION TO REGULAR EXPRESSIONS

- A good knowledge of regular expressions is indispensable to solving problems related to string processing and that includes spam filtering.
- Chapter 4 of my book “Scripting with Objects” explains in great detail how to use regular expressions in Perl and Python scripts. [\[If you do not have the book, you might at least want to look at the scripts in the book that are online.\]](#)
- [The regular expression engine that is now used by a large number of languages is the one that was first developed for Perl.](#) This is the engine that is used by Python, Java, C++ based packages, etc. [Unfortunately, this is not the same engine that is used by Procmail, the main utility used for spam filtering in Unix/Linux based platforms.](#) Fortunately, the regular expressions as used in Perl/Python, on the one hand, and as used by Procmail, on the other, have much in common. Additionally, by what is known as Condition Line Filtering, you can always ask Procmail to send any email to a Perl/Python based script for processing. So in the remainder of this section, we will focus mainly on the regular expressions that can be used

with Perl and Python. [Procmail uses what are known as Unix regular expressions. For information on the regex engine used by Procmail, do either ‘man regexp’ or ‘man egrep’.]

- To become proficient with regular expressions, you must learn:
 - How to use **anchor metacharacters** to force matching to take place at line and word boundaries
 - How to use **character classes** to specify alternative choices for a single character position in the matching process
 - How to specify **alternative subexpressions** inside a regular expression
 - How to use **grouping metacharacters** to extract substrings from a string
 - How to use **quantifier metacharacters** to control repetitions in a string
 - The difference between **greedy** and **non-greedy** quantifier metacharacters
 - How to use **match modifiers** to force matching to be, say, case-insensitive, global, etc.
 - More advanced topics in regular-expression based processing include **non-capturing groupings**, **lookahead** and **look-behind** assertions, etc.
- String processing with both Perl and Python harnesses, on the one hand, the power of regular expressions, and, on the other,

the support provided by the language's I/O facilities, control structures, and so on.

- A regular expression helps search for desired strings in text files **under very flexible constraints**, such as when looking for a string that starts with a particular sequence of characters and ends in another sequence of characters without regard to what is in-between. [Through a regular expression, one can also specify the location of the substring to search for in relation to the beginning of a line, the end of a line, the beginning of a file, etc. Further constraints that can be built into a regular expression include specifying the number of repetitions of a given elemental pattern, whether the matching of the regular expression with an input string should be greedy or non-greedy, etc. Regular expressions are also useful in search-and-replace operations in text processing, for specifying the separators for splitting long strings of text substrings, etc.]
- We will refer to the string that will be subject to regex matching as the *input string*. [This is simply a device to make it easier to differentiate between the different strings involved in regex examples. The input string will often be read one line at a time from a text file, which justifies *input* in the name *input string*. But an input string may also be specified directly in a program.]
- The script `word_match.pl` shown below, taken from Chapter 4 of my SwO book, illustrates the basic syntax of using Perl's match operator `m//` for regular expression matching. Our regular expression in this case is the string `hello`. The script will ask you to enter strings in the terminal window in which

you execute this script. Each string you enter will be matched with the regular expression pattern. If the match is successful, the script will print out the portion of the input string before the match, after the match, etc.

```
#!/usr/bin/perl -w

## word_match.pl

use strict;
my $regular_expression = "hello";
print "Enter a line of text:\n";
while (chomp( my $input_string = <> ) ) {
    if ( $input_string =~ /$regular_expression/ ) {
        print 'The line you entered contains "hello"', "\n";
        print "The portion of the line before the match: ", $', "\n";
        print "The portion of the line after the match: ", $', "\n";
        print "The portion of the line actually matched: ", $&, "\n";
        print "The current line number read by <>: ", $., "\n";
        print "\nEnter another line of text or Ctrl-C to exit:\n\n";
    } else {
        print "\nNo match --- try again or enter Ctrl-C to exit\n\n";
    }
}
```

- The regular-expression based matching in the above script takes place in the conditional of the **if** statement:

```
$input_string =~ /$regular_expression/
```

where `=~` is the Perl's **binding operator**. In the syntax shown above, the two forward slashes, `'//'`, which delimit the regular expression, are a shorthand for `'m//'`, the Perl's **matching operator**.

- Shown below is a Python version of the `word_match.pl` script. This is also from Chapter 4 of my SwO book:

```
#!/usr/bin/env python

## word_match.py
## works with both Python 2.x and Python 3.x
import re
regular_expression = r'hello'
while 1:
    import sys
    try:
        if sys.version_info[0] == 3:
            input_string = input("\nEnter a line of text: ")
        else:
            input_string = raw_input("\nEnter a line of text: ")
    except IOError as e:
        print(e.strerror)
    m = re.search( regular_expression, input_string )
    if m:
        # Print starting position index for the match:
        print( m.start() )
        # Print the ending position index for the match:
        print( m.end() )
        # Print a tuple of the position indices that span this match:
        print( m.span() )
        # print the input strings characters consumed by this match:
        print( m.group() )
    else:
        print("no match")
```

- Note that the regular-expression based matching in the Python script is carried out by the statement:

```
m = re.search( regular_expression, input_string )
```

The call `re.search()` returns an object of type `MatchObject`.

The rest of the code then extracts the needed information from this object. [Regular expression matching in Python is carried out with the `re` module.

Also note that the prefix `r` for a string argument causes all the characters in the string to be accepted literally.]

- In both the Perl and the Python examples shown above, we used a simple pattern, `hello`, as our regular expression. The matching functions invoked in both scripts looked for this pattern *anywhere* in the input string.
- But if you wanted to see if the input string contained a pattern at, say, just the beginning, or at just the end? Now your regular expression would need to use what are known as *anchor metacharacters*.
- Perl and Python use the same set of metacharacters. Typically, you'd want the match to take place either at the very beginning of the input string, or at the very end. The anchor metacharacter `^` is used to force a match to take place at the beginning of the input string and the anchor metacharacter `$` to force the match to take place at the end of the input string. [The regex `^abra` will match the string `abracadabra`, but not the string `cabradababra`. Similarly, the regex `dabra$` will match the string `abracadabra`, but not the string `dabracababra`. In addition to forcing a regex match to take place at the beginning and the end of a line with the help of anchor metacharacters, it is also possible to force a regex to match at the beginning or the end of a word boundary. Both Perl and Python use the anchor metacharacter `\b` to denote the word

boundary. The symbol `\b` can stand for both a non-word to word transition and a word to non-word transition. So the regex `\bwhat` will match the string `whatever will be will be free`, but not the string `somewhat happier than thou`. Similarly, the regex `ever\b` will match the string `whatever will be will be free`, but not the string `everywhere I go you go`. Note that the anchors do not consume any characters from the input string during the matching operation.]

- We will now talk about **character classes for regex matching**. When we specify a regex as, say, `hello`, a successful match between this regex and an input string requires the input string to possess exactly the same sequence of characters wherever the match is scored.
- What if we want more than one choice for an input-string character for a given character position in a regex? Suppose we want to detect for the presence of the following substrings in an input string:

`stool spool skool`

Can we specify a single regex for extracting all three substrings? Yes, we can do so with the help of a *character class*. For example, the regex `s[tpk]ool` which includes the character class `[tpk]` will be able to search for any of the three words `stool`, `spool`, and `skool`.

- A character class is simply a set of choices available for a specific character position in a regex. The most general notation for a character class calls for placing the set of choices inside square brackets. The expressive power of a character class can be enhanced by using special characters; **these are metacharacters that have specifically designated meanings inside the square-bracket notation** for a character class.
- For both Perl and Python, these *character-class metacharacters* are

- ^] \

The character class metacharacter ‘-’ acts like a range operator for a character class. It allows a compact notation for a character class consisting of a sequence of either alphabetically contiguous characters or numerically contiguous characters. For example, the character class `[a-f]` is simply a more compact way of writing `[abcdef]` and the character class `[3-9]` is a more compact of writing the `[3456789]` pattern.

- Here are some other illustrations of the use of the range operator inside a character class:

regex	matches with
----- var[0-9]	----- var0, var1, var2,, var9

<code>[0-9a-fA-F]</code>	a digit or letter in a hex sequence
<code>[nN] [oO] [pP] [eE]</code>	nope, NOPE, Nope, etc.

- The character-class metacharacter '-' loses its special meaning if it is either the first or the last character inside the square brackets.
- Let's now talk about ^ as a character-class metacharacter. If this character is the first character inside the square brackets, it negates the entire character class. What that means is that any input-string character except those in the character class will be acceptable for matching:

regex	matches with
-----	-----
<code>[^0-9]</code>	will match any non-digit character
<code>[^a-zA-F]</code>	will match any non-alphabetic character
<code>[^c]at</code>	will match aat, bat, dat, eat,

If the character ^ appears anywhere except at the beginning of a character class, it loses its special meaning vis-a-vis the character class. *Note that a negated character class does not imply a lack of character at that position in the input string.*

- Let's now talk about **specifying alternative subexpressions in a regex**. It is sometimes necessary to specify a list of alternatives for one or more portions of a regex. For example, if **Joe** and **Mary** would work out equally for a job and you want to see if an

input string mentions either name, you could specify a regex as the `\bJoe|Marry\b` pattern. The operator `|` is usually called the ‘or’ operator. If it is possible that **Joe**’s name could also show up as **Joseph**, we could incorporate that possibility in our regex by rewriting it as the `\b(Jo(e|seph))|Marry\b` pattern.

- When there exist alternatives in a regex for scoring a match with an input string, the regex engine seeks the earliest possible match and, as soon as the engine is successful, stops trying out any remaining alternatives *even if one of the remaining alternatives provides what seems like a ‘better’ match*. In the following example:

```
input_string = "hellosweetsie"
regex = h(ey|ello|i)(sweet|sweetsie)
```

Only the “hellosweet” portion of the input string will be used to score a successful match with the regex, even though it would seem that all of the input string would provide a ‘better’ — in the sense of being a more complete — match.

- Note that when a match with the input string does not work out with the first choice in a set of alternatives, backtracking is used to try each of the remaining choices. [To explain why we use the word ‘backtracking’ to describe the matching process in the presence of alternatives, let’s say we have

two alternatives in the first portion of a regex and two alternatives in the remaining portion. Let's also say we have a successful match between the input string and the first of the two alternatives in the first portion of the regex. But, then, we are not able to match either of the two alternatives in the second part of the regex with what remains of the input string. Now the matcher must backtrack and try the second choice in the first portion of the regex.]

- We will now talk about using **parentheses for grouping subexpressions** in a regular expression. In addition to being used for specifying alternatives, as you have already seen, parentheses can also be used to return input string groupings that match specific subexpressions in a regex. When used for grouping, the parentheses are known as the *grouping metacharacters*. [A pair of matching parentheses surrounding a subexpression creates a unit for the following purposes: (i) For specifying one of multiple choices, as you saw earlier. (ii) For being subject to repetition through the use of quantifier metacharacters. (iii) For extracting a desired substring from an input string. The input-string substring that matches a parenthesized portion of a regex is available to the rest of the program through a special variable. It is also available inside later portions of the regex through a *backreference*. (iv) For specifying non-capturing groupings in regexes. Non-capturing parentheses have special notation — '(?:)' — as oppose to '()'. (v) For specifying lookahead and lookbehind assertions. The parentheses are used in the form '(?=)' for lookahead assertions and '(?<=)' for lookbehind assertions.]
- Consider the following example of an input string and a regex:

```
input string  =  hellothere! how are you
regex        =  (hi|hello)there
```

The regex engine stores in a special variable the input-string substring that matches a parenthesized portion of a regex. Perl actually stores such a substring in two separate variables, one available in the regex itself and the other available outside the regex in the rest of the program. Let's first focus on the variables available in the rest of the program that allow us to extract the input-string portions that matched a parenthesized subexpression in a regex. These variables, called *matching variables*, are named:

`$1 $2 $3 $4`

The value of `$1` is set to the input-string substring that matches the first parenthesized subexpression in a regex, the value of `$2` to the substring that matches the second parenthesized subexpression, and so on. The same substrings from the input string are available inside a regex through the *backreferences*:

`\1 \2 \3 \4`

- What Perl achieves with matching variables is accomplished in Python by calling the `group()` method on a match object. If `m` denotes the match object returned by a call to `re.search()`, `m.group(1)`, `m.group(2)`, etc., will return portions of the input string that match with the parentheses-delimited subexpressions of the regex. The backreferences work the same in both Perl and Python — as demonstrated by the Python script that follows the next Perl script.

- Before showing you the scripts with examples of matching variables and backreferences, note that Perl and Python also allow us to specify *nonextracting groupings* or *noncapturing groupings*. The non-capturing version of '()' is '(?:)'. That is, you attach the symbol pair '?:' to the left parenthesis.
- Shown below is a Perl script, taken from Chapter 4 of my book SwO, that demonstrates how we can extract the portions of an input string that match a regex. The extracted portions are shown in the commented-out sections.

```
#!/usr/bin/perl -w

## Grouping.pl

use strict;

# Demonstrate using match variables:
my $pattern = 'ab(cd|ef)(gh|ij)';           #(A)
my $input_string = "abcdij";               #(B)
$input_string =~ /$pattern/;               #(C)
print "$1 $2\n";                          # cd ij          #(D)

# Demonstrate the binding op returning a list of
# matched subgroupings:
$pattern = '(hi|hello) there(,|!) how are (you|you all)'; #(E)
$input_string = "hello there, how are you.";             #(F)
my @vars = ($input_string =~ /$pattern/);                #(G)
print "@vars\n";                                         # hello , you   #(H)

# Demonstrate using backreferences:
$pattern = '((a|i)(l|m))\1\2';                       #(I)
@ARGV = '/usr/share/dict/words';                       #(J)
while (<>) {                                           #(K)
    print if /$pattern/;                               #(L)
}
# output of while loop:
```

```
# balalaika
# balalaikas
```

- Shown below is a Python version of the Perl script shown above. This one is also from Chapter 4 of SwO.

```
#!/usr/bin/env python

### Grouping.py

import re # (A)

# Demonstrate using group() for extracting matched substrings:
pattern = r'ab(cd|ef)(gh|ij)' # (B)
input_string = "abcdij" # (C)
m = re.search( pattern, input_string ) # (D)
print( m.group(1), m.group(2) ) # cd ij # (E)

# Another demonstration of the above:
pattern = r'(hi|hello) there(,|!) how are (you|you all)'; # (F)
input_string = "hello there, how are you."; # (G)
m = re.search( pattern, input_string ) # (H)
print( m.group(1), m.group(2), m.group(3) ) # hello , you # (I)

# Demonstrate using backreferenes:
filehandle = open( '/usr/share/dict/words' ) # (J)
pattern = r'((a|i)(l|m))\1\2' # (K)
done = 0 # (L)
while not done: # (M)
    line = filehandle.readline() # (N)
    if line != "": # (O)
        m = re.search( pattern, line ) # (P)
        if ( m != None ): # (Q)
            print(line) # (R)
        else: # (S)
            done = 1 # (T)
filehandle.close() # (U)
# output of while loop:
# balalaika
# balalaikas
```

- Let's now talk about using **quantifier metacharacters in regular expressions**. A *quantifier metacharacter* is used to control the number of repetitions of the immediately preceding smallest possible subexpression in a regex.
- Both Perl and Python use the following as quantifier metacharacters:

* + ? {}

A quantifier metacharacter is placed immediately after whatever portion of the regex it is that we want to see repeated.

- The metacharacter '*' means an indefinite, including zero repetitions of the preceding portion of the regex. The regex 'ab*' will match the following input strings

a
ab
abb
abbb
abbbb
...
...

It is obviously straightforward to interpret the behavior of the quantifier '*' when it applies to a single preceding character (that is not a metacharacter), as in the above example where it is applied to the character 'b'.

- But now let's examine the pattern '**a[bc]***' as a regex where the quantifier '*' now applies to the character class '**[bc]**'. It is best to visualize this regex as a shorthand way of writing a whole bunch, actually an indefinitely large number, of the following regexes:

```
a
a[bc]
a[bc] [bc]
a[bc] [bc] [bc]
a[bc] [bc] [bc] [bc]
...
...
```

- If there exists a match between the input string and any of these indefinitely large number of regexes, the regex engine will declare a successful match between the input string and the regex.
- Now consider the subexpression '**.***' that is used very commonly in regexes. Let's say our regex is the '**a.*b**' pattern. This regex is a compact way of writing an indefinitely large number of regexes that look like

```
ab
a.b
a..b
```

a...b

a....b

a.....b

and so on

Any input string that matches any of these regexes would be considered to be a match for the regex.

- The quantifier metacharacter ‘+’ again means an indefinite repetitions of the preceding subexpression as long as there is at least one occurrence of the subexpression.
- When a part of a regex is followed by the quantifier metacharacter ‘?’, that means that the subexpression is an optional part of the larger regex, meaning that it can appear zero or one times.
- If it is desired to specify the number of repetitions at the both the high end and at the low end, one can use the quantifier metacharacters ‘{ }’. The regex, for example, ‘a{n}’ where ‘n’ is a specific integer value means that exactly ‘n’ repetitions of ‘a’ are allowed. Therefore, the regex ‘a[bc]{3}’ is a short way of writing ‘a[bc][bc][bc]’ as a regex.
- A variable number of repetitions within specified bounds is

expressed in the following manner: ‘**a{m,n}**’ where ‘m’ and ‘n’ are specific integer values, the former specifying the minimum number of repetitions of the preceding subexpression and the latter the maximum number.

- The quantifier metacharacters we have shown so far are greedy, in the sense they gobble up as much of the input string as possible. For some string matching problems, you need what are known as **non-greedy quantifiers**. The *non-greedy quantifiers* are also known as *minimal-match* quantifiers. The non-greedy version of the greedy quantifiers ***** **+** **?** **{}** are ***?** **+****?** **??** **{}****?**, respectively.
- So, as far as the notation is concerned, the non-greedy version of each quantifier is the corresponding greedy version with ‘?’ attached as a postfix. As with ‘*’, the quantifier ‘*?’ stands for an indefinite number of repetitions of the preceding subexpression in the regex, but it will choose as few as possible.
- Let’s now talk about **match modifiers**. The matching of a regular expression with a string can be subject to what are known as *match modifiers* that control various aspects of the matching operation.
- The modifier flags themselves are not directly a part of a regex. They are more a language feature and, therefore, how they are

specified is different in Perl and Python.

- For **case insensitive matching**, Perl uses the modifier `//i`. And in Python you need to supply the option `re.IGNORECASE` to the matching function.
- Ordinarily the regex stops at the first possible position in the input string where there is a match with the regex. But if you want the regex engine to continue chugging along and scan the entire input string for all possible positions where there exist matches with the regex, you have to set the **global option** as a **match modifier**. The match modifier in Perl for the global option is `m//g`. In Python you have to call the function `re.findall()`.
- What precisely is returned by the regex engine when you set the global option depends on two factors: (i) whether or not the regex contains any groupings of subexpressions; and (ii) the evaluation context of matching.
- All of our discussion so far has dealt with input strings that consisted of single lines, which were either read one line at a time from an input file or were specified directly so in the program. **Another match modifier is to take care of the case when the input string consisting of multiple lines.**

[Back to TOC](#)

31.8 USING procmail FOR SPAM FILTERING

- As mentioned previously, Procmail is a mail processing utility for Unix. When used for controlling spam, a procmail filter is applied at the MDA level. In other words, a procmail filter is applied BEFORE an email goes to your MUA. (See Section 31.2 for what the acronyms MDA and MUA mean.)
- The first version of **procmail** was written in 1991 by Stephen R. van den Berg. But now its maintenance is supervised by Philip Guenther. Procmail is open source.
- A lot of information about procmail can be gleaned from the following manpage commands in Unix or Linux:

```
man procmail
man procmailrc
man procmailsc
man procmailex  (A very useful manpage for recipe examples)
```
- A procmail filter will be invoked by your local MDA if you include the following sort of a line in your **.forward** file

```
"|/usr/local/bin/procmail #kak"
```

where **you must replace** 'kak' by your own login name. If you are outside the 'ecn' domain at Purdue, you must also replace the path to the **procmail** utility with what it is on the host where the MTA to MDA transfer of email takes place. The pipe symbol at the very beginning of the string in the **.forward** file tells the Sendmail program to make the email available to the Procmail program on its standard input. What follows '#' is really a comment that **sendmail** may use to make your **.forward** file unique in its own cache.

- The very first thing that Procmail does is to look for the file

\$HOME/.procmailrc

in your home directory. The email is processed according to the **recipes** laid out in the **.procmailrc** file. If no **.procmailrc** file can be found or if the *processing of the email according to the recipes in .procmailrc reaches the end of the file without any resolution*, Procmail stores the email in the default system mailbox for your account, which for me would be **/var/mail/kak** on RVL4. [Included in the code that you can download from the

lecture notes web site is a file called **dot_procmailrc**. You can use it as your starter **.procmailrc** file. Make sure you change the name of the file from **dot_procmailrc** to **.procmailrc**]

- A **.procmailrc** file consists of three parts:

1. Assignment of relevant environment information to local variables

2. Assignments to variables that will be used locally as macros in the `.procmailrc` file

3. Recipes

- Here is the beginning portion of my `.procmailrc` file:

```
SHELL=/bin/sh
PATH=/usr/local/lib/mh:$PATH
MAILDIR=$HOME/Mail
LOGFILE=$HOME/Mail/logfile
#VERBOSE=1
VERBOSE=0
EOL="
"
LOG="$EOL$EOL$EOL"
LOG="New message log:$EOL"
LOG='perl GET_MESSAGE_INDEX'
LOG="$EOL"
```

where `SHELL`, `PATH`, `MAILDIR`, and `LOGFILE` are local variables that store the environment information needed by Procmail. The variables `VERBOSE` and `EOL` are the two other local variables; the first controls the level of detail placed in the log files and the second defines the end-of-line character for log entries. The variable `EOL` defines a macro that can subsequently be used through the `$EOL` syntax shown in the last line. Note that all these variables are local to the `.procmailrc` file. **Any assignment to the local variable `LOG` generates information that is written to the logfile.** Note the call `'perl GET_MESSAGE_INDEX'` for

associating an integer index with each entry in the logfile. The Perl script `GET_MESSAGE_INDEX` merely reads an integer value stored in a local file, increments that integer, uses it for the current entry in the logfile, and writes the incremented value back to the file where the index is stored. In this manner, you can associate an integer index with each entry in the log file — something that comes in handy if you want to see quickly how many emails your spam filter has processed so far. [Included in the code that you can download from the lecture notes web site is the `GET_MESSAGE_INDEX` script file that I use.]

- I will now talk about the third part of a `.procmailrc` file — the part consisting of recipes. A recipe in a `.procmailrc` file will ordinarily consist of the following three parts:

1. A colon line (always begins with `:0` for historical reasons)

```
:0 [flags] [ : [locallockfile] ]
```

I will have more to say about the ‘`flags`’ and ‘`locallockfile`’ through illustrations of the colon line that you will soon see.

2. A condition (or conditions) starting in a new line. A condition line always begins with a ‘`*`’. There can be only one condition per line. However, you can have any number of condition lines.

Everything in a condition line after ‘`*`’ is processed by the `egrep` regex engine. [As previously mentioned, for information on the regex engine used by Procmail, do either ‘`man regex`’ or ‘`man egrep`’.] Any white space between the ‘`*`’ that marks the start of a condition-line and the first non-blank character that comes after that in the same line is ignored.

Multiple conditions, each in a different condition line, are “anded” together. No condition lines mean “true” by default.

3. An action starting in a new line. **There can only be one action line in a recipe.**

- Shown below is a recipe that is meant for trapping an email that contains even a single non-English or non-numeric character in its subject line. Note that because of the action line `/dev/null`, the action consists of deleting such emails.

```
:0 :
* ^Subject.*[^ [:alnum:][:punct:]]+.*$
/dev/null
```

where the metacharacters `^` and `$` carry the same meanings as described in Section 31.7. Note the use of the character classes `[:alnum:]` and `[:punct:]`. These are defined for the `egrep` regex engine; the first stands for the English alphanumeric characters (it is the same as the character class `[0-9A-Za-z]`), and the second stands for the punctuation marks.

- Here are some examples of the colon line. The examples also illustrate the use of flags in the colon line. **Note that when there is a second colon present in the same line, as in the second recipe, a local lockfile is used to properly sequence the processing of emails should they arrive much too quickly.** That

is, should a new email arrive while the previous one is being processed by a recipe with a lockfile indicator, the new email will be made to wait until the previous one has exited the recipe.

- :0 The simplest case. Only the header is egrep'd, meaning that only the header is sent to the regex engine.
- :0 : The second colon causes a local lockfile to be used if multiple emails arrive concurrently.
- As this recipe is being used, its invocation for the next email if it arrives at about the same time will be put on hold.
- Important only if you are writing to a file.
- :0 B The recipe will be applied only to the body of the email
- :0 H The recipe will be applied only to the headers. This, by default, is the same as the first case shown above.
- :0 HB The recipe will be applied to both the head and the body
- :0 c a copy of the email will be processed by this recipe; the original email will continue to be processed by the remaining recipes.
- :0 D Tell the internal egrep to be case-sensitive in matching regexes in the condition lines. The default is case insensitive.

:0 f This sends the email to the program named after the pipe symbol in the action line. Procmail expects the external program to return a modified email on the standard input. Further processing by procmail is then carried out on this modified email. THIS FLAG CREATES FILTERING RECIPES.

:0 fhw You will use this for a filtering recipe that tells procmail that the body of the email will NOT be changed by the external filtering program. In other words, the external program in the action line will only change the header of the email. All that is accomplished by the 'h' flag. The 'w' flag tells procmail to wait for the filtering program to return and TO CHECK THAT IT EXECUTED SUCCESSFULLY.

.... and many others (see procmailrc manpage)

- The following characters immediately after '*' in a condition line have special meaning. You can think of them as Procmail *condition line metacharacters*.

! Invert the condition.

? Use the exit code of the specified program
(This is called CONDITION LINE FILTERING)

< Check that the total length of email is less
than the number of bytes that is specified after
this character

> Opposite of above

and others (check procmailrc manpage)

- Here are examples of simple recipes:

Recipe 1:

```
:0:
* ^From.*joe.shmoe
* ^Subject.*seminar.(announce.*|notice)
junkMail

# Recipe 2:
:0:
* !^From.*groothuis
* ^From.*root
junkMail

# Recipe 3:
:0:
* ^From.*joe.*bureaucrat
* ^To.*engfaculty
junkMail

# Recipe 4:
:0 HB:
* ^Content-Type: text/html
* !(charset="?us-ascii"?|charset="?iso-8859-1"?)
junkMail

# Recipe 5:
:0 HB
* ^Content-Disposition:.*attachment
* < 300000
{
  :0 c
  ! avi_kak@yahoo.com

  :0 c:
  medium_attachments

  :0 :
  /var/mail/kak
}
```

- You will find two kinds of recipes in the list shown above:

Delivering Recipes: These cause the email to be written to a file, or to be forwarded to another email address, or to be absorbed by a program. **Procmail quits processing the email when it encounters a delivering recipe.** Recipes 1 through 4 in the list shown above are delivering recipes.

Non-delivering Recipes: These are recipes that cause the output of a program to be captured back by Procmail. The procmail then continues processing this new output in the same way it processes as a regular email. **A non-delivering recipe is also used to start a *nested block* of recipes.** Recipe 5 shown on the previous page is a non-delivering recipe.

- As shown by the nested block in Recipe 5 above, a delivering recipe can be made to behave like a non-delivering recipe by specifying the “c” flag in the colon line. The “c” flag stands for “copy”. This causes a copy of the email to be sent to the delivering recipe while the original is saved for processing by the rest of the .procmailrc file.
- **The sole action line** that is allowed in a recipe starts with one of the following symbols:

!	the email is forwarded to the email address that comes after this symbol
	the email is piped into the program you name after this symbol

```
{      this marks the beginning of a nested block of
      recipes; the block must end in a matching '}'

none of the above  ----  whatever is in the action line
                        is taken to be the name of a
                        mailbox file in which the email
                        is deposited.
```

You saw the first (‘!’), the third (‘{’), and the fourth of four action-line possibilities in the five recipes shown earlier.

Note the very different roles played by the character ‘!’ in a condition line and in an action line.

- I will now talk about **condition line filtering** in recipes. For condition line filtering, the condition line must have the character ‘?’ after the mandatory ‘*’ character at the beginning of the line. Consider the recipe:

```
:0 HB:
* < 15000
* ? $MAILDIR/condfilter2.pl 2>&1
junkMail
```

This recipe feeds the email into the Perl script **condfilter2.pl**. The condition succeeds if the Perl script returns the exit code of 0 and fails if the exit code returned is 1. The string ‘2>&1’ redirects the **STDERR** stream to the **STDOUT** stream (which the filtering program redirects into the log file).

- I will now show a simple example of condition line filtering. The name of the Perl script shown below is `condfilter2.pl`. This is the script that is called in the second condition statement in the recipe shown above. The main job of this script is to first construct a single string from all of the Base64 encoded material that forms a single multimedia partition in the email and to then invoke the `decode_base64()` function from the `MIME::Base64` module on the encoded string in order to decode it. Then if the size of this decoded string is less than a threshold, an email to considered to be potential spam. [It might seem strange that we would want to declare an email to possibly be spam merely on the basis of the size of its Base64 decoded attachment. But note that such a filter would be invoked only AFTER a lot of other tests that would have declared the message to be non-spam if that was indeed the case. Base64 encoding is commonly used by spammers to hide their text content.]

```
#!/usr/bin/perl -w
use strict;

use MIME::Base64;

my $encoded_string = "";
my $decoded_string = "";
my $content_html_flag = 0;
my $encoding_flag = 0;

open LOG, ">> /home/rvl4/a/kak/Mail/log_condfilter2";

# Change default for output from STDOUT to LOG. Since this is
# a condition line filter, its actual output is not of any use
# to procmail. Procmail only needs to know whether the program
# exits with status 0 or a non-zero status.
select LOG;

print "\n\n";          # separator for new log entry

while ( <STDIN> ) {
    chomp;
    if ( /^From:/ ) {
        print "$_\n";
        next;
    }
}
```

```

    if ( /^Date:/ ) {
        print "$_\n";
        next;
    }
    if ( /content-type.*text\/html/i ) {
        $content_html_flag = 1;
        next;
    }
    if ( $content_html_flag && /content.*encoding.*base64/i ) {
        $encoding_flag = 1;
        next;
    }
    next if $content_html_flag == 0;
    next if /^Content-T/;
    next if /^X-/;
    next if /\s*$/;
    $encoded_string .= $_;

    last if ( /\s*$/ && ( $encoded_string ne "" ) );
}
if ( $encoding_flag == 0 ) {
    print "Exited with non-zero status because no text/html content.\n";
    print "This e-mail will stay in processing stream.\n";
    exit(1);
} else {
    $decoded_string = decode_base64( $encoded_string );
    my $length = length( $decoded_string );
    print "length of the decoded string: $length\n";
    if ( $length < 15000 ) {
        print "Exited with status 0 because of short base64-encoded\n";
        print "content. Potential spam\n";
        print "This e-mail will go to junkMail.\n";
        exit(0);
    } else {
        print "text/html encoded content is large. Possible not spam.\n";
        print "Exited with non-zero status.\n";
        print "This e-mail will stay in the processing stream of procmail.\n";
        exit(1);
    }
}
}

```

- I will now talk about **filtering recipes**. A filtering recipe merely modifies the email, [but keeps it in the processing pipeline for the recipes that follow](#). The example shown below only modifies the ‘Subject:’ line in the header:

```

:0
* ^From.*ack

```



```

* ^Subject.*the key is[ ]+\/*[0-9a-z].*
{
  KEY='echo $MATCH | sed 's/[^0-9a-zA-Z]//g' | tr 'A-Z' 'a-z' '
  SUBJECT='echo "the key you supplied $KEY"'

  :0 fhw
  | formail -I "Subject: $SUBJECT"

  :0
  !kak@purdue.edu
}

```

To understand this recipe, you must know about the special role played by the symbol pair ‘\/' in the second condition line.

Whatever portion of the subject line in the email being processed by this recipe matches the regex that comes after ‘\/' becomes implicitly the value of the local variable `MATCH`. Next we have a local variable `KEY` inside a sub-recipe. Because of the backquotes, the value of `KEY` will be whatever is returned by the Unix process in which the command(s) that is/are within the backquotes is/are executed. The first Unix command is `echo`; this command simply echos its argument to the standard output, where it is picked up by the second Unix command `sed`, etc. What that means is that the string value of the local variable `MATCH` will be subject to a modification by the `sed` command, and so on.

- To explain further the syntax of the assignment to the local variable `KEY` at the top of the nested recipe shown in the

previous bullet:

```
KEY='echo $MATCH | sed 's/[^0-9a-zA-Z]//g' | tr ' [A-Z]' '[a-z]' '
```

The command `sed` as invoked here accepts the characters on its standard input and drops all non-alphanumeric characters.

Therefore, it can also get rid of any spaces that the email might have in the key value in the subject line. The output of `sed` is piped into the Unix utility `tr` that simply carries out a

‘translation’ from uppercase to lowercase. The output of `tr` is written to the standard output, where it is captured by the backticks operator, and the output of the backticks operator becomes the value of the local variable `KEY`. [The assignment statement shown

above is just to illustrate how you can invoke various Unix/Linux utilities inside a recipe. You may or may not want to use the `sed` and `tr` utilities in the manner I have shown.]

- Also note that I am using the Unix/Linux utility `formail` to modify the `Subject` header of the email. The ‘`-I`’ option to `formail` will cause any existing `Subject` fields in the email processed to be deleted before inserting the new such header. For a further explanation of what else happens in the above filtering recipe, see the explanations that follow since I have used the same example below.
- I will next show a small recipe file called `my_recipe_file` whose job is to accomplish the following:

```
-- to trap incoming email from the 'ack' account
```

```

-- to extract the 'Subject:' header of the incoming
   mail, especially the part that comes after the
   phrase 'the key is'

-- to extract the 'Date:' header of the incoming
   email

-- to insert a new 'Subject:' header for the outgoing
   email

-- to insert a new 'Date:' header for the outgoing
   email

-- and, finally, to insert some additional text just
   after the headers in the outgoing email.

```

Here is what is in the file `my_recipe_file`:

```

# name of this file: my_recipe_file

SHELL=/bin/sh
MAILDIR=$HOME/proc_folder
LOGFILE=$HOME/proc_folder/logfile
#VERBOSE=1
VERBOSE=0
EOL="
"

LOG="$EOL$EOL New message log:$EOL"

:0
* ^From.*ack
* ^Subject.*the key is[ ]+\/*[0-9a-z].*
{
  KEY='echo $MATCH | sed 's/[^0-9a-zA-Z]//g' | tr ' [A-Z]' '[a-z]','
  SUBJECT='echo "the key you supplied $KEY"'
  DATE='formail -x Date:'

:0
{
  :0 fhw
  | formail -I "Subject: $SUBJECT"

:0 fhw

```

```

    | formail -I "Date: $Date"
}

:0 fhw
| cat -; echo "<><><>MESSAGE AT THE BEGINNING OF NEW BODY<><><>"

:0
!kak@purdue.edu
}

```

- In the recipe shown shown above, note the following two different uses of the `formail` Unix utility. I first use this utility in the line:

```
DATE='formail -x Date:'
```

This invokes the `formail` program in a separate process on account of the backticks that you see in the line. The backticks will cause `formail` to read data on its standard input and to output the results on the standard output. Whatever `formail` returns becomes the value of the variable `DATE` in the `procmail` program. The `-x` option extracts the “Date” field from the header of the email read from the standard input.

- Now note the second different use of `formail` in the action line for the recipe shown in the file `my_recipe_file`:

```
formail -I "Subject: $SUBJECT"
```

Here I am using `formail` to insert the `Subject:` header in the email being compose by the filtering recipe. As mentioned previously, the `-I` option will cause the previous value of the “Subject” header to be replaced by the new value.

- So whereas the first use of `formail` is extracting information from the incoming email, the second use is inserting information into the email being composed for output.
- In the file `my_recipe_file`, note the condition line

```
* ^Subject.*the key is[ ]+\/*.*[0-9a-z].*
```

As mentioned earlier, everything that gets consumed by that part of the regex that comes after `\/*` is deposited in the Procmail variable `MATCH`. Therefore, if the `Subject:` header of the incoming message is something like

```
Subject: the key is AbcDEF 123
```

the string ‘AbcDEF 123’ will become the value of the local variable `MATCH`.

- Again in the file `my_recipe_file`, notice from the following action line how I am adding some additional text to the body of the incoming email to form the body of the outgoing email:

```
| cat -; echo "<><><>MESSAGE AT THE BEGINNING OF NEW BODY<><><>"
```

The `echo` function will place in the standard output the text that is given to it as the argument. This additional text will appear BEFORE the body of the incoming email because only the flag ‘h’ is in the colon line of this sub-recipe. Regarding the invocation ‘`cat -`’, note that the basic job of the command

cat is to send to standard output whatever it reads from its argument. When the argument is just the symbol ‘-’ the command **cat** takes its input from whatever the standard input happens to be. In our case, the recipe would send to the standard input the header of the incoming email. So, in the example shown above, the **cat** command will simply redirect the header to the standard output, where it is subsequently followed by the output of the **echo** command. It is this mechanism that causes the argument to **echo** to be placed just after the email header.

- The previous case showed the following sub-recipe for inserting a message at the beginning of email

```
:0 fhw
| cat -; echo ‘<><><>MESSAGE AT THE BEGINNING OF NEW BODY<><><>’
```

We could also have used

```
:0 fbw
| echo ‘<><><><>MESSAGE AT THE BEGINNING OF BODY<><><><>’; cat -
```

In the first case, the ‘h’ flag is crucial; and in the second case, the ‘b’ flag is crucial. The ‘h’ flag makes available only the header section on the standard input. The ‘b’ flag makes available only the body at the standard input. [Recall that the ‘-’ argument to **cat** causes the standard input to be used for reading the input. Of course, in both cases, **cat** will make its output available at the standard output.]

- I should also point out that for experimenting with a recipe, you do NOT have to put it in a `.procmailrc` file at the top level of your home directory. **For testing purposes, your recipe can be in any file in any directory.** For example, the recipe file `my_recipe_file` that I showed earlier could be tested in any directory with a command line like:

```
procmail my_recipe_file < mail_file
```

where the file `mail_file` is some file that contains a previously collected email message for testing purposes.

[Back to TOC](#)

31.9 HOMEWORK PROBLEMS

1. Your ability to write procmail recipes for trapping spam depends entirely on your proficiency with regular expressions. To figure out for yourself how good you are at constructing regular expressions, can you create an example for each of the eleven regex related items shown in magenta on page 27?

2. Programming Assignment:

Using the “starter kit” made available through the Lecture 31 code link at Lecture Notes website, design a `procmail` based spam filter that would trap all 75 messages in the `junkMail.tar.gz` gzipped tar archive. When you gunzip and untar the archive with, say,

```
tar -zxvf junkMail.tar.gz
```

you’ll see 75 individual spam messages with names `junkMail_1` through `junkMail_75`. About these messages:

junkMail_1 through junkMail_50 : The headers of all these messages have one thing in common: they contain multiple entries in the “From:” header. All these messages were trapped by a single recipe in your instructor’s spam filter. The regex in your instructor’s recipe has only 40 characters

in it. (If the regex engine used by procmail allowed for Perl's '{ }' metacharacters, this regex could have been made as short as just 10 characters.)

junkMail_51 through junkMail_63 : These messages can be trapped just on the basis of the “Subject:” line in the email headers.

junkMail_64 through junkMail_66 : In your instructor's spam filter, these messages were trapped on basis of the content (email body) of the messages.

junkMail_67 through junkMail_75 : You can trap these with a single recipe that contains compound rules. Here is an example of a recipe with compound rules:

```
:0 HB:
* ^Content-Type: text/plain
* !^Content-Type: text/html
* !^content-type: application/pdf
* !^content-type: application/zip
* !^content-type: application/msword
* !^content-type: application/*.signature
* Content-Transfer-Encoding: base64
junkMailCompound6
```

What this says is that if the “Content-Type” MIME header is `text/plain` and none of the MIME objects are of type PDF, ZIP, etc., and yet the “Content-Transfer-Encoding” MIME header calls for Base64 encoding, then there is a great chance it is a spam message. By the way, this is the NOT

the compound recipe you need for trapping the messages `junkMail_67` through `junkMail_75`.

After you have incorporated the new recipes in your `.procmailrc` file, you can test your filter on an individual message by invoking the command:

```
procmail .procmailrc < junkMail_XX
```

where “XX” is the integer suffix for the message file. Obviously, you would need to write either a shell script, or a Python script, or a Perl script to execute the above command in a loop for all 75 spam messages. If your recipes work on all 75 messages, you will not see any messages being subject to the default action of your procmail filter, which is usually to put the surviving messages in your mailbox `/var/mail/account_name`.

Since the spam messages in the tar archive are in their raw form, it is sometimes difficult to see what is in them — especially if the MIME objects in the messages are Base64 encoded. To help you decipher those spam messages that are fully or partially encoded, you’ll find in the starter kit a Perl script named `EmailParser2.pl`. Execute this script and give it a command-line argument that is the name of the junk mail file you want to decipher. It will deposit the different MIME objects in the email in a subdirectory called `mimemail` in the directory in which you execute the script.