1 Operating System Security

In this lab we are going to examine file permissions and a few recent bugs in software that have cause major security concerns.

1.1 Hidden Directories

Try making a directory named "test" with the `mkdir test` command.

Now rename that directory to "test2" with the `mv test/ test2/` command.

Type `ls` to view the files and directories in your current folder. Note that "test" was renamed to "test2".

Next make a hidden directory by renaming the directory one last time to ".test" with the `mv test2/ .test/` command.

Run `ls` and note that a ".test" folder is not listed.

Navigate into the hidden folder by running the `cd .test/` command. You are now inside a hidden directory.

Go back up a directory by running the `cd ..` command.

Run the command `ls -la`. The `-l` flag lists items line by line and the `-a` flag lists all files and directories (including hidden files and directories). Note that these "hidden" files or directories are not really a security mechanism because access is not restricted (security through obscurity).

Hidden files are commonly used for storing user preferences or preserving the state of a utility. Usually the intent of hidden files is simply to reduce "clutter" in the display of the contents of a directory listing with files the user did not directly create.

1. After running `ls -la` in any directory you will see two directories named "." and "..". What are these directories and why are they in every directory?

1.2 File Permissions

In your current directory type `ls -l`. The `l` option tells the `ls` program to list each file/sub directory as a line. Notice that each line looks something like:

```
-rw-r--r--   1 bholland  staff  5954 Mar  6 12:33 README.md
drwxr-xr-x   7 bholland  staff   238 Apr 15 00:25 lecture39
drwxr-xr-x   9 bholland  staff   306 Apr 17 08:08 lecture40
drwxr-xr-x   8 bholland  staff   272 Apr 19 16:50 lecture41
drwxr-xr-x   7 bholland  staff   238 Mar  6 12:33 lecture42
```

You can now see the file permissions associated with each item. Each file and directory has three user based permission groups: owner, group, and everyone (all users). Each file or directory also has three basic permission types: read, write, and execute.

Each line of output resulting from the `ls -l` command is displayed as: `_rwxrwxrwx 1 owner:group`. The `_` is a special permission flag that is typically used to denote whether the item is a directory (represented with the character `d`) or a file (represented with the character `-`). The three sets of `rwx` stand for read, write, and execute permissions being granted for the owner, group, and all users permission groups (respectively). If the permission is not granted then a `-` character is substituted in place of the permission type. The last piece of information is the owner and group assignment.

For more information read https://www.linux.com/learn/tutorials/309527-understanding-linux-file-permissions.

Before answering the questions below you may want to read the man pages for the following programs `chmod`, `chown`, and `chgrp`.

2. What command would you type to find the owner of a file named "secrets.dat"?

3. What command would you type to change the owner of a file named "secrets.dat" to your username?

4. What would be the effect of running the command "`chmod -R 777 .`"? Why would you want to be very careful when running this particular command?

1.3 Shellshock (Bash bug)

By some estimates the Shellshock bug has gone undiscovered for nearly 25 years. There are some good explanations of the bug online.

Shellshock Code and the Bash Bug - Computerphile

5. Describe the Shellshock bug and how it works.

The impact of the Shellshock bug was originally under estimated. The impact of the bug became much more clear as many researchers realized that several programs (including popular web servers such as Apache) make heavy use of environment variables.

Example: http://www.securitysift.com/shellshock-targeting-non-cgi-php/

6. What is an environment variable and how could it be used in conjunction with the Shellshock bug to remotely exploit a web server?

1.4 Buffer Overflows

The basic buffer overflow example we have looked at in class is shown below.

```c
#include <stdio.h>
int main(int argc, char **argv) {
  char buf[64];
  strcpy(buf, argv[1]);
}
```

Remember that it is vulnerable because:

- The program is soliciting input from the user through the program arguments
- Input is stored to memory (buf)
- Input bounds are not checked and data in memory can be overwritten
- The main function has a return address that can be overwritten to point to data in the buffer

Year after year, buffer overflows have ranked among the worst offenders when it comes to serious security issues. Programmers are slowly learning to code more securely and there are many tools that can analyze your code and point out security issues during development. With all the modern technology trying to eliminate buffer overflows, you might be wondering how they keep working their way into our code!

Take a look at the infamous Sendmail Crackaddr bug below. This code also contains a buffer overflow. Can you spot it? If you need help check out the extra resources below.

```
# define BUFFERSIZE 200
# define TRUE 1
# define FALSE 0
int copy_it ( char * input , unsigned int length ) {
  char c, localbuf [ BUFFERSIZE ];
  unsigned int upperlimit = BUFFERSIZE - 10;
  unsigned int quotation = roundquote = FALSE ;
  unsigned int inputIndex = outputIndex = 0;
  while ( inputIndex < length ) {
    c = input [ inputIndex ++];
    if ((c == '<') && (! quotation )) {
      quotation = TRUE ;
      upperlimit --;
    }
    if ((c == '>') && ( quotation )) {
      quotation = FALSE ;
      upperlimit ++;
    }
    if ((c == '(') && (! quotation ) && ! roundquote ) {
      roundquote = TRUE ;
      // upperlimit --; // decrementation was missing in bug
    }
    if ((c == ')') && (! quotation ) && roundquote ) {
      roundquote = FALSE ;
      upperlimit ++;
    }
    // if there is sufficient space in the buffer , write the character
    if ( outputIndex < upperlimit ) {
      localbuf [ outputIndex ] = c;
      outputIndex ++;
    }
  }
  if ( roundquote ) {
    localbuf [ outputIndex ] = ')';
    outputIndex ++;
  }
  if ( quotation ) {
    localbuf [ outputIndex ] = '>';
    outputIndex ++;
  }
}
```

The bug was discovered in 2003 by Mark Dowd. Later Thomas Dullien extracted a smaller toy example shown above. The original flaw was contained in about 500 lines of code and the toy example is about 50 lines of code. Finding the bug in this code is difficult enough with a single loop, but the original bug contains about 10 loops (with nested loops of depth of 4), GOTOs, lots of pointers, pointer arithmetic, and calls to string functions. Most program analysis tools a good at finding surface level bugs like the examples in class, but this tool is nested much "deeper" in the code. Finding these types of bugs is an ongoing research project.

Resource: Sendmail Crackaddr Talk

Resource: A Java version of the same bug (throws an Index out of Bounds Exception)

7. Explain how an attacker can trigger the buffer overflow in the Sendmail Crackaddr Toy example.

1.5 Closing Thoughts

Automatic Exploit Generation (AEG) an exciting new field that uses program analysis to automatically analyze programs for exploitable features and then automatically generates and tests proof of concept exploits. This field has wide implications for security at both the industrial and nation-state levels. We are living in exciting times for computer security, and its future is in your hands!