Assignment 2
CS-456
Fall 2015

Due Date: 11/26/2015 at 11:59 pm (No extensions)
This Project May be Done in Groups of 4

Goals:

1. To extract binary codes from executable files and convert them into hexadecimal format.
2. To implement a binder of executable files which combines multiple executable files into a single executable file; a technique often used by attackers.
3. To assess the impacts of binder programs on malware distribution underground economy.

WARNING!!! PLEASE READ BEFORE CONTINUING!!!

This assignment does not involve implementation of self-propagating malware and therefore can be safely done on any **any 32-bit Ubuntu system**. However, this assignment **shall be graded using 32-bit Ubuntu 12.04 system**. It is in your best interest to test your submission on such system prior to submitting it.

If you choose to do this assignment on your own Ubuntu 12.04 system, you will need to install g++ version 4.8. Please see the following link for the instructions for installing g++ version 4.8.

Overview

In class we learned about control hijacking attacks and binders of executable files. Implementing these requires understanding of the techniques used for manipulating and combining binary codes. These skills are important for developing vulnerability patches, executing and countering control hijacking attacks, analyzing malware, and in many other facets of network security.

This assignment has two parts. In the first part you will experiment with a primitive technique which can be used for hiding malicious codes on Microsoft Windows systems. In the second part, you are going to implement a program for combining multiple binary executables into a single executable file which when ran executes all the constituent executables. This type of program is called a *binder*.

Part I

In order to complete this part you will need a Windows 7 VM which you can find on your security lab node.

1. Boot your VM.
2. Download the http://www.7-zip.org compression program.

3. Launch the installer and follow the installation instructions.
4. Download the `worm.bat` file from one of the earlier demos given in class. The file can be found on Titanium. Recall, the file illustrates a very simple Windows worm ***which can hang your system***.
5. Right-click on `worm.bat`, and from the menu choose `7-zip` → `Add to worm.7z`
6. Find a `.gif` image.
7. Copy the `.7z` and `.gif` files to the same directory (if you have not already done so).
8. Hold the `Shift` key and right-click in the directory containing the two files.
9. From the drop-down right-click menu choose: `Open command window here`.
10. In the terminal window that appears, enter the following command: `copy /B <gif file name> + <7z file name> result`. For example, `copy /B mygifile.gif + myzipfile.7z result`.
11. The above command should have created a file called `result` in the same directory.
12. Rename the `result` file to `result.7z` (i.e., append the `7z` extension).
13. Try opening the file using the 7-zip program. What happens? (Note: one way to open the file using the 7-zip program is to right-click on `result.7z` and choose `7-zip` → `Open archive`. What happens? Are you able to extract and run the `worm.bat` file inside the archive?
14. Repeat the above steps, but this time rename the file to `result.gif` extension.
15. Try opening the file. What happens?

1. Explain what is happening. Do some research in order to find out what the above `copy` command does. In your explanation be sure to explain the role of each argument in the above command. Also, be sure to explain how Windows handles files which leads to the above behavior. Include the answers to these questions in the README file you submit.
2. How can this technique be used for hiding malicious codes?
3. How robust is this technique in terms of avoiding detection by anti-virus tools? You may need to do some research.

Part II

In this part of the assignment, you are going to implement an executable binder program. The program shall take names of multiple executable files as command line arguments and merge these executables into a single executable file called `bound`. Executing the `bound` binary shall execute all of the constituent executables.

The binder program shall be invoked using `python binder.py` $\langle\ PROG_1\ \rangle\ \langle\ PROG_2\ \rangle \ldots \langle\ PROG_N\ \rangle$ command line where each $PROG_i$ is an executable program. For example, `python binder.py /usr/bin/ls /usr/bin/pwd`.

The program architecture comprises two main components:

1. A python script called `binder.py`: this script has the following flow of logic:

   (a) Call the `hexdump` program on each executable file provided as command line argument in order to extract and convert the binary contents of each executable file into a sequence of C/C++-style, 1-byte hexadecimal values (i.e. each value comprising two hexadecimal digits prefixed with `0x`. For example, `0x1a`). Before proceeding with this part, you should spend sometime experimenting with the `hexdump` utility on the terminal. Try the following sequence of commands: `hexdump /usr/bin/ls`. What is the output? What does this output: `"hexdump", "-v", "-e", ’"0x" 1/1 "%02X" ","’, /usr/bin/ls`. You will find this command useful in this assignment. More information about hexdump utility can be found on http://manpages.debian.org/cgi-bin/man.cgi?query=hexdump&sektion=1.

   (b) Generates a C++ header file called `codearray.h`. This file contains a C/C++-style array of pointers to unsigned characters `codeArray`. The pointer at index location i of the array points to the array of hexadecimal numbers representing executable instructions and data of each executable file previously extracted using the `hexdump` utility. For example, lets assume we have three programs. The programs have sizes of 5, 3, and 6 bytes respectively. The `codeArray` should be defined as

   ```
   unsigned char* codeArray[3] =
   {new char[5]{0x43, 0x91, 0xa1, 0xdb, 0xff},
   new char[3]{0x12, 0xfe, 0xab},
   new char[6]{0x56, 0x45, 0x8d, 0x37, 0x78, 0x30}
   };
   ```

   In the same file, the script adds an array of integers `programLenghts` where each index location i contains the length of the i^{th} row of `codeArray`. It also adds a C/C++-style macro or a constant `NUM_BINARIES` specifying the number of programs to be bound. For example, assume we have three programs from the previous example. The script will add array `programLenghts[3] = {5, 3, 6}`; and macro `#define NUM_BINARIES`.

   (c) Finally, the script invokes the `g++` compiler program in order to compile the `binderbackend.cpp` file which includes the `codearray.h` as header file and executes the binary codes in the `codeArray` array defined therein. The g++ compilation line is `g++ binderbackend.cpp -o bound -std=gnu++11`.

2. A C++ program called `binderbackend.cpp` which is a program implementing the logic for executing the codes of each executable in the array `codeArray` declared and initialized in `codeArrays.h`. This program, when executed, has the following flow of logic: for each row i in the `codeArray`:

   (a) Generate a randomly named temporary file. This can be achieved using the `tmpnam()` function which will create a randomly named file in the `/tmp` directory. The syntax for calling `tmpnam()` is `char* fileName = tmpnam(NULL);`

   (b) Give the file executable permissions; can be done using `chmod(fileName, 0777)`;

   (c) Write all columns of row i to the above-create file. This can be achieved using the `fwrite()` function; please see the `fwrite.cpp` sample file for details. Please note: doing this operation requires us knowing the number of elements in each row. This value we can find in the `programLenghts` array at index location i.

(d) Issue a `fork()` system call which will spawn another process which is a clone of the process that has issued `fork()` system call. The new process is called a child process while the original process is called a parent process. Both processes will continue executing with the next line following the `fork()` system call. In the parent process, the `fork()` system call returns the process id of the child, while in the parent it returns 0.

(e) The child process shall invoke the `execlp()` system call in order to turn into process which runs an executable file we just created, while

(f) The parent process repeats the above steps on row $i + 1$ until all rows of the matrix have been processed. You can find an example on how to use `fork()` and `execlp()` system calls in `fork.cpp`. Also, the following link gives a nice tutorial on the same.

(g) The parent process then waits for all children to finish.

The skeleton files for `binder.py` and `binderbackend.cpp` have been provided. Please check out the `TODO:` comments.

BONUS:

Implement the same binder program on the Windows platform.

Grading guideline:

- Program executes when run 10'
- Correct answers to part I and submission of the file `result` containing an image file merged with a zip file. 15'
- Correct implementation of the binder program. 70'
- README file: 5'
- Bonus: 15'
- Late submissions shall be penalized 10%. No assignments shall be accepted after 24 hours.

Academic Honesty:

Academic Honesty: All forms of cheating shall be treated with utmost seriousness. You may discuss the problems with other students, however, you must write your **OWN codes and solutions**. Discussing solutions to the problem is **NOT** acceptable (unless specified otherwise). Copying an assignment from another student or allowing another student to copy your work **may lead to an automatic F for this course**. Moss shall be used to detect plagiarism in programming assignments. If you have any questions about whether an act of collaboration may be treated as academic dishonesty, please consult the instructor before you collaborate. Details posted at http://www.fullerton.edu/senate/documents/PDF/300/UPS300-021.pdf.