Exercise 3+4 – More advanced shellcodes.

README.1ST:

1. Download the exercise pack from the course website, and use the ex_unpack utility to open it.
2. **The installation fixes a bug from exercise_mode.py from ex02, if you haven't run the old exercise_mode.py please run the new one now found under tools. If you did run it, please restart your VM.**
3. A new shellcode studio – You will find a new set of binary under 'tools' directory for this exercise
    a. shellcode_host – reads a binary shellcode as instructed via the command line, and simulates execution.
    b. shellcode_host_no_nulls – similar to shellcode_host, but the string is copied via strcpy, so no null characters (0x00) will be permitted in the body of the shellcode.
    c. stack_overflow_host – similar to shellcode_host in the sense that it will allow null bytes inside the shellcode, but here you must overflow the stack and control the return address yourself.
    d. stack_overflow_host_no_nulls – similar to stack_overflow_host, but no null bytes will be permitted
    e. Each file comes complete with source code, a standard binary, and a binary containing debugging symbols ("_sym" postfix).
    f. When you're asked to provide a shellcode to run on any of the above, make sure you supply a version (or multiple versions) that can work on both versions of the binaries you were supplied.
4. New shellcode compilation tool – under the tools directory, look for the build_shellcode.py script (it is based on the patch_util_gcc.py script, but is made for simpler usage when creating shellcodes).
5. Please include all .asm .bin and .txt files for submission of this exercise, for each question, please submit a q[0-9].txt which includes details of your work/thought process in your own words.

Questions -

0. Q0 – basic code execution:
    a. Use the following tools from the tools directory:
        i. 'example_shellcode.asm'
        ii. build_shellcode.py script
        iii. shellcode_host
    b. Build the shellcode and run it inside the new environment:
        i. "./build_shellcode.py example_shellcode.asm q0.bin"
        ii. ./shellcode_host q0.bin
    c. See that you managed to get code execution and a shell. If you have any problems – please talk to the instructions.

1. Q1 – write a reverse-shell code:
    a. Build a setup just like you did in Q0, but this time – make the shellcode do a different task – make it connect back via TCP and run a shell you can interact with over the network.
    b. You'll need to –
        i. Create a new socket with the socket() function
        ii. Build a struct to describe the target (you may want to build it in C and copy the data)
        iii. Call the connect() function to connect 'home'
        iv. On the receiving end, you can use your own tcp server (c / python) or more easily – use the netcat program – "nc –l –p [PORT]" (The port will be the last 4 digits of your ID#).
        v. Now that you have connected, you can use the dup2() function to set your stdin, stdout and stderr file descriptors (values 0, 1 and 2) to point to the new socket.
        vi. Finally, call the exec syscall just like we did before, and run /bin/sh
        vii. You should now have an interactive shell connected to the netcat 'server'
    c. Since this is a complex shellcode, work in stages and try to use gdb to debug your work. You may want to write your code in C first, and compare / copy parts where possible.
    d. Make this shellcode work with shellcode_host_no_nulls, by avoiding any null bytes in your code.
2. Q2 – Remote code execution:
    a. You'll find the code and binaries (with and without symbols) for the 'network_daemon' program. Answer in q2.txt , q2.asm, q2.bin.
    b. It is a simple network server, that supports director listing (like 'ls' / 'dir') and file read operations.
    c. You will also find the network_client.py program, which can be used to explore the server functionality. You will also find some documentation about the request format in the client code.
    d. You may connect to the server by running the command: "./network_client.py 127.0.0.1 7331"
    e. First, locate any major vulnerabilities in the network_daemon code. And document your findings in in q2.txt.
    f. Now, adapt the basic shellcode to work in this scenario, by modifying the client code to exploit the vulnerabilities you just found. q2f.py
    g. Once this is successful, make your new shellcode work in this scenario as well. Don't forget to add as many NOPs as you can to make it resilient.
3. Q3 – Firewall proof remote code execution:
    a. We will now rewrite the shellcode, to work with the existing connection file descriptor. (On the same executable from Q2). Don't forget to document everything you do with q3.txt,q3.asm,q3.bin)
    b. In order to do see, we must first locate the existing socket fd on the stack.
    c. We are then only left with simple task of fitting the code from Q2 to fit this scenario.

d. Run the new shellcode, you are now firewall proof.

4. Q4 – Polymorphic shellcode:

   You will write polymorphic shellcode that does not require any character whose ascii ordinal is larger or equal to 0x80. There are two parts to the shellcode (decoder) and the actual shellcode. [document all work in q4.txt, provide q4.asm, q4.py]

   i. Decoder:
   ii. Map out all the bytes in your code where you have chars >= 0x80.
   iii. Now write asm code that given the address to itself in eax in the beginning modifies the values of the itself in pre-specified locations by using XOR on byte ptr with bl that contains 0xFF
   iv. Find a way to store 0xFF in bl without using 0xFF. There are many ways.
   v. Write a python script that appends the code you just wrote to shellcode from example_shellcode.asm (q0.bin) automatically, so that it creates a new shellcode without chars larger than 0x80.
   vi. To test it, hard code the address to eax in the beginning and run with shellcode_host. **From here on out it's a bonus worth 15pts.**
   vii. Look at the executable at Q2, Now use pop eax (or pop ax if necessary for alignment) to make eax contain the last valid EBP on the stack.
   viii. Use inc eax/dec eax, to make eax point to the beginning of the code.
   ix. Since you can not use NOP for your slide use "\x48\x40" (inc eax; dec eax).
   x. Make everything work together.
   xi. Make it work with the shellcode from Q3. **[5pts bonus]**