Buffer Overflow Demo (DIY)

Here is DIY demo you can perform in the comfort of your own home. It is the same demo as in the class. It may help you with the assignment as well (it's a bit of a duplicate)

Included Files:

vuln.c – vulnerable source code

shellcode.txt – list of inputs to use

Instructions: (note: $ and (gdb) are command prompts, not input)

1. First you need to disable an overflow defense mechanisms. Execute the following command:

   ```
   $ sudo sysctl -w kernel.randomize_va_space=0
   ```

   This disables a randomization process that makes addresses in the stack unpredictable.

2. Compile vuln.c

   ```
   $ gcc -o prog vuln.c -fno-stack-protector –z execstack
   ```

   The fno-stack-protector flag removes other run-time overflow protections.

3. The program takes in an argument and copies it into a buffer. We know the size of the buffer is 1024, so we can just try a larger number and overflow it.

   ```
   $ ./prog `perl -e 'print "A"x1050'`
   ```

   This perl command creates a string of 1040 A's to feed in as an argument. The program should generate a segmentation fault.

4. Use gdb to get a better idea of what happened.

   ```
   $ gdb prog
   (gdb) run `perl -e 'print "A"x1050'`
   ```

   This generates a segmentation fault. Let's look at the register values to see what happened

   ```
   (gdb) i r
   ```

   Notice that we overwrote the eip and ebp values with 0x41414141 which is "AAAA." This tells us that we were able to overwrite the return value instruction pointer. Here's what memory looked like after we overflowed the buffer:

   ```
   |AAAAAAAAAAA...AAAAA|AAAA|AAAA|A...
   |   Buffer          |ebp |eip |other memory
   ```

   After we returned from the function the stored ebp and eip values were moved back to the registers as we saw.

5. Now we need to calculate how many bytes we really need to precisely overflow the return

address (stored eip). We know the buffer is 1024, the ebp is 4 bytes, and the eip is 4 bytes. There is also an exception frame buffer of 8 bytes that is added to the stack before local variables. So, if we had an argument of 1040 bytes, that should work. Repeat step #4 with that amount and see what happens. Make it a byte smaller and see what happens to the eip register value.

6. The next step is to get some shellcode that will spawn our command prompt. Just use the shellcode in input.txt. We need to know how big the shellcode is so we can calculate our input size appropriately. If you count up the bytes (/xx is one byte) you should get 38. Then you need to add the string at the end where each character is a byte (/bin/sh has 7 bytes because we count the slashes). So the shellcode is 45 bytes long.
7. Besides the shellcode we need an input of size 995 (1040 – 45). We will tag the shellcode at the end of the input, but remember, we want to overwrite the return address with a specific 4-byte value. So our input string will now have 991 bytes, followed by our 45-byte shellcode, followed by our 4-byte return address.
8. In gdb run the program again with the following argument:

   ```
   (gdb) run `perl -e 'print "A"x991,"<insert shellcode>","BBBB"'`
   ```

   If it worked you should be able to look at the registers and see that eip is now 0x42424242, or "BBBB."
9. We need change the "BBBB" to be an address that will allow us to execute our shellcode. It is difficult to identify the exact address where the shellcode starts, so we'll use a "nop-sled." This is a string of no-operation instructions. Execute the program in gdb again, replacing "A" with "\x90". Now you'll have 991 nop instructions followed by the shellcode. You only need to point anywhere in the nop-sled to get the shellcode to execute.
10. Now we need to find an address in the nop-sled. After executing the program in gdb with the new input, execute the following command:

    ```
    (gdb) x/2000xb $esp
    ```

    This will list 2000-bytes of memory starting with the top of the stack. Look for the nop-sled. Select an address within the boundaries of the sled. Assume we find an address 0xa1b2c3d4. Change the "BBBB" in the input string to "\xd4\xc3\xb2\a1".
11. After executing the program with the new input you should receive a command prompt.

Summary:What we were able to do is overflow a buffer and replace the return address (the address of the next instruction to be executed after the function returns) with something else. We replaced with an address in the range of our nop-sled when lead to the execution of our shellcode.

Modification:What would happen if you changed prog to be a setuid program? Try it and see.