Lecture 22: Buffer Overflow Attack

Lecture Notes on "Introduction to Computer Security"

by Avi Kak (kak@purdue.edu)

May 25, 2007

©2007 Avinash Kak, Purdue University

Goals:

- Services and ports

- A Case Study for Computer Security: The `telnet` Service

- Some Security Bulletins Concerning the `telnet` Service

- Buffer Overflow Attack: Understanding the Call Stack

- Overrunning the Allocated Memory in a Call Stack

- Demonstration of Program Misbehavior Because of Buffer Overflow

- What Other Damage Can Buffer Overflow Cause

Services and Ports

- Every service on a machine is assigned a port. On a Unix/Linux machine, the ports assigned to standard services are listed in the file **/etc/services**. Here is a very small sampling from this list from my Linux laptop:

```
# Network services, Internet style
#

# The latest IANA port assignments can be gotten from
# http://www.iana.org/assignments/port-numbers
# The Well Known Ports are those from 0 through 1023.  The Registered
# Ports are those from 1024 through 49151 The Dynamic and/or Private
# Ports are those from 49152 through 65535

# Each line describes one service, and is of the form:
#
#service-name  port/protocol  [aliases ...]    [# comment]

echo 7/tcp
echo 7/udp
daytime 13/tcp
daytime 13/udp
ftp-data 20/tcp
ftp 21/tcp
ssh 22/tcp # SSH Remote Login Protocol
telnet 23/tcp
smtp 25/tcp mail
time 37/tcp timserver
domain 53/udp
whois++ 63/tcp
tftp 69/tcp
finger 79/tcp
http 80/tcp www www-http # WorldWideWeb HTTP
kerberos 88/tcp kerberos5 krb5 # Kerberos v5
hostname 101/tcp hostnames # usually from sri-nic
```

```
pop3 110/tcp pop-3 # POP version 3
sunrpc 111/tcp portmapper # RPC 4.0 portmapper TCP
sunrpc 111/udp portmapper # RPC 4.0 portmapper UDP
auth 113/tcp authentication tap ident
auth 113/udp authentication tap ident
sftp 115/tcp
sftp 115/udp
uucp-path 117/tcp
nntp 119/tcp readnews untp # USENET News Transfer Protocol
ntp 123/tcp
netbios-ns 137/tcp # NETBIOS Name Service
imap            143/tcp          imap2           # Internet Mail Access Protocol
ipp 631/tcp # Internet Printing Protocol
rsync 873/tcp # rsync
imaps   993/tcp # IMAP over SSL
pop3s 995/tcp # POP-3 over SSL
biff 512/udp comsat
login 513/tcp
who 513/udp whod
shell 514/tcp cmd # no passwords used
printer 515/tcp spooler # line printer spooler
printer 515/udp spooler # line printer spooler
talk 517/udp
router 520/udp route routed # RIP
uucp 540/tcp uucpd # uucp daemon

netstat 15/tcp # (was once asssigned, no more)
............
............
and many many more, see /etc/services for the complete list.
```

- It is important to note that when we talk about the services on a machine, it does not necessarily imply that the agent using that service on a remote machine is a human. In fact, many of the services running on your computer are for the benefit of other computers (and other devices such as printers).

- The running computer program that provides a network service is frequently called a **daemon server** or just a **daemon**.

A Case Study for Computer Security: The `telnet` Service

- Let's now consider the telnet service in particular since it has been the subject of a fairly large number of security problems. **Telnet is the industry standard protocol that allows a user to establish a remote terminal session on a remote machine. The session gives a user the ability to execute commands on the remote machine. Until not too long ago, especially before the advent of SSH, telnet was the commonly used method to gain access to remote machines on the internet for terminal session. For example, if you wanted to log into, say, `moonshine.ecn.purdue.edu` from your personal machine, you would use the command 'telnet `moonshine.ecn.purdue.edu`'.)**

- From the table shown on Slide 2, the constantly running `telnetd` daemon monitors port 23 for incoming connection requests.

- When a client seeks such a connection, the client runs a program called `telnet` that sends to the server machine a **socket number**, which is a combination of the IP address of the client machine together with the port number that the client will use for communicating with the server. When the server receives the client **socket number**, it acknowledges the request by sending back to the client its own socket number.

- Starting with the next slide, let's now look at some of the security bulletins that have been issued with regard to the telnet service.

Some Security Bulletins Concerning the `telnet` Service

- On February 10, 2007, US-CERT (*United States Computer Emergency Readiness Team*) issued the following Vulnerability Note:

  ```
  Vulnerability Note VU#881872

  OVERVIEW:  A vulnerability in the Sun Solaris telnet daemon (in.telnetd)
  could allow a remote attacker to log on to the system with elevated
  privileges.

  Description: The Sun Solaris telnet daemon may accept authentication
  information vis the \verb+USER+ environment variable. However, the
  daemon does not properly sanitize this information before passing it
  on to the login program and login makes unsafe assumptions about the
  information.  This may allow a remote attacker to trivially bypass the
  telnet and login authentication mechanisms. .....

  This vulnerability is being exploited by a worm .....
   ......
   ......
  ```

  The problem occurs (supposedly because of the buffer overflow attack) if you make a connection with the string "`telnet -l -froot`".
  (As a side note, US-CERT (`http://www.us-cert.gov/`) was established in 2003 **to protect the internet infrastructure. It publishes Vulnerability Notes at** `http://www.kb.cert.org/vuls/`**.)**

- As mentioned in the Vulnerability Note, there is at least one worm

out there that can make use of the exploit mentioned above to break into a remote host either as an unprivileged or a privileged user and execute commands with the privileges of that user.

- On December 31, 2004, CISCO issued the following security advisory:

```
Cisco Security Advisory: Cisco Telnet Denial of Service Vulnerability

Document ID: 61671
Revision 2.4

Summary:

A specifically crafted TCP connection to a telnet or a reverse telnet
port of a Cisco device running Internetwork Operating System (IOS) may
block further telnet, reverse telnet, remote shell (RSH), secure shell
(SSH), and in some cases HTTP access to the Cisco device.  Data Link
Switching (DLSw) and protocol translation connections may also be
affected.  Telnet, reverse telnet, RSH, SSH, DLSw and protocol
translation sessions established prior to exploitation are not affected.
....
....
This vulnerability affects all Cisco devices that permit access via
telnet or reverse telnet.......
....
....
Telnet, RSH, and SSH are used for remote management of CISCO IOS devices.
```

- On February 7, 2002, Microsoft released the following security bulletin:

```
Microsoft Security Bulletin MS02-004
```

Problem: A vulnerability exists in some Microsoft Telnet Server products
            that may cause a denial-of-service or allow an attacker to
            execute code on the system.

Platform: Telnet Service in Microsoft Windows 2000

Damage: A successful attack could cause the Telnet Server to
            fail, or in some cases, may allow an attacker to execute
            code of choice on the system.

.....
.....

Vulnerability Assessment: The risk is HIGH. Exploiting this
                            vulnerability may allow an attacker
                            complete control of the system.
Summary:

    Unchecked buffer in telnet server could lead to arbitrary code
    execution.
    ....
    ....
    The server implementation ..... contains unchecked buffers in
    code that handles the processing of telnet protocol options.

    An attacker could use this vulnerability to perform buffer
    overflow attack.
    ....
    ....
    A successful attack could cause the Telnet server to fail, or
    in some cases, could possibly allow attackers to execute code
    of their choice on the system.
    ....
    ....
    The vulnerability exists because of an unchecked buffer in
    a part of code that handles the Telnet protocol options.
    By submitting a specially specific malformed packet, a
    malicious user could overrun the buffer.
    ....
    ....

Buffer Overflow Attack: Understanding the Call Stack

- Let's first look at the two different ways you can allocate memory for a variable in a C program:

```
int data[100];

int* ptr = malloc( 100 * sizeof(int) );
```

The first declaration will allocate memory on the stack at compile time and the second declaration will allocate memory on the heap at run time. (**Of course, with either declaration, you would be able to use array indexing to access the individual elements of the array. So,** `data[3]` **and** `ptr[3]` **would fetch the same value in both cases, assuming that the same array is stored in both cases.**)

- **Buffer overflow** occurs when information is written into the memory allocated to a variable on a stack **but the size of this information exceeds what was allocated at compile time.** *(It is also possible to create buffer overflows in a heap; those are usually known as "heap buffer overflows". Basically, you can create an overflow any time information is placed in a memory block assigned to an array without checking the bounds on what is actually placed there.)*

- So to understand the buffer overflow attack, you must first understand how a process uses its stack. What we mean by a **stack** here is also referred to as a **run-time stack**, **call stack**, **control stack**, **execution stack**, etc.

- When you run an executable, it is run in a process. As the process executes the main function of the program, it is likely to encounter calls to various subroutines inside the main function, subroutines inside the called subroutines, and so on. A process keeps track of the sequence of the subroutines called and local variables encountered along the way with the help of a **call stack**.

- Consider the following C program:

```
// ex.c

int main() {
    int x = foo( 10 );
    printf( "the value of x = %d\n", x );
    return 0;
}
int foo( int i ) {
    int ii = i + i;
    int iii = bar( ii );
    int iiii = iii;
    return iiii;
}
int bar( int j ) {
    int jj = j + j;
    return jj;
}
```

- Let's now focus on what is in the call stack for the process in which the program is being executed at the moment when `foo` has just called called `bar` and the statement 'int jj = j+j' of `bar()` has just been executed. Although the precise details regarding how a call stack would look like depend on the machine architecture and the spcific compiler used, the following is not an unrealistic model for the assembly code generated by the `gcc` compiler for Intel x83 architectures:

```
stack_ptr-->   jj                                    ^
               retrun address to caller              | stack frame for bar
               j                                      V

               iii                                    ^
               ii                                     | stack frame for foo
               return address to caller               |
               i                                      V

               x                                      ^
               argc                                   | stack frame for main
               argv                                   V
```

Note that the call stack consists of a sequence of stack frames, one for each calling function that has not yet finished execution. In our case, `main` called `foo` and `foo` called `bar`. The top stack frame is for the function that just got called and that is currently being executed. The memory locations within the stack are accessed with the help of a **stack pointer** that always points to the top of the stack.

- The local variables at the top in each stack frame are those that

are still in scope at the current moment. That is why the stack frame for **foo** shows the variable **iii** at the top and not yet the variable **iiii**, since the latter has not yet been seen (when **bar** was called).

- As each local variable is encountered, it is pushed into the stack. That is why the variable **jj** is at the top of the stack. Subsequently, as each variable goes out of scope, it is popped off the stack. In our simple example, when the thread of execution reaches the right brace of the body of the definition of **bar**, the variable **jj** would be popped off the stack and what will be at the top will be pointer to the top of the stack frame for the calling function **foo**.

- How the stack is laid out can be seen by looking at the assembly code. For the C program that we showed above, we could generate an assembly code file by using the '-S' option for the **gcc** command:

```
gcc -S -O ex.c -o ex.S
```

where we have turned on optimization with the '-O' flag in order to remove unnecessary assembly-level instructions. Shown below is a section of the assembly output in the file **ex.S**:

```
...     .....   .....
...     .....   .....
foo:
```

```
        pushl   %ebp
        movl    %esp, %ebp
        subl    $20, %esp
        movl    8(%ebp), %eax
        sall    %eax
        pushl   %eax
        call    bar
        leave
        ....    ....
        ....    ....
.globl bar
.type bar, @function
bar:
        pushl   %ebp
        movl    %esp, %ebp
        movl    8(%ebp), %eax
        sall    %eax
        leave
        ret
```

- To see what the above assembly output says about the call stack layout, note that the Intel x86 *calling convention* (which refers to how a calling function passes parameters values to a called function and the former receives the returned value) uses three three 32-bit registers that hold the following pointers:

Stack Pointer: The name of the register that holds this pointer is **esp**, the last two letters referring to "stack pointer". This register always points to the top of the process call stack.

Base Pointer: This register is denoted **ebp**. This register is used to reference the function parameters and the local vari-

ables in the *current* stack frame. This pointer is also frequently called the **Frame Pointer**.

Instruction Pointer: This register is denoted **eip**. This holds the address of the next CPU instruction to be executed.

- Shown below is the annotated version of the assembly output that shows more clearly the construction of the call stack:

```
...    .....    .....
...    .....    .....
foo:
    pushl      %ebp               push the value in register ebp into stack
    movl       %esp, %ebp         move the value in esp to ebp (we are using
                                  the AT&T (gcc) syntax:  'op source dest')
    subl       $20, %esp          subtract decimal 20 from value in esp
                                  (so stack ptr will now point to 20 locations
                                   down, meaning in the direction in which
                                   the stack grows as you push info into it)
    movl       8(%ebp), %eax      move to accumulator a value that is
                                  decimal 8 + the value stored at the
                                  memory address pointed to by ebp
                                  (this moves local var i into accumulator)
    sall       %eax               shift left (i+i means 2*i means shift left)
                                   thus executing i+i of foo
    pushl      %eax               push accumulator value into stack
                                  (result saved on stack is now the var ii
                                   where it can serve as argument to bar)
    call       bar                call bar
    leave
    ....       ....
    ....       ....
.globl bar                        (directives useful for assembler/linker
        .type  bar, @function      begin with a dot)
bar:
    pushl      %ebp               push the value of register ebp into stack
```

15

```
movl        %esp, %ebp          copy the value of esp to ebp
movl        8(%ebp), %eax       copy 8 plus value in epb into accumulator
sall        %eax                shift left the accumulator
leave
ret
```

- Note that by convention the stack grows downwards (which is opposite from how a stack is shown pictorially) and that, as the stack grows, the addresses go from high to low. So when you push a 4-byte variable into the stack, the address to which the stack pointer will point will be the previous value minus 4. This should explain the **sub** instruction (for subtraction). The 'l' suffix on the instructions shown (as in **pushl**, **movl**, **subl**, etc.) stands for 'long', meaning that they are 32-bit instructions. (**By the same token, the suffix 'b' stands for single byte instructions, and 'w' for 'word', meaning 16-bit instructions.**)

Buffer Overflow Attack: Overrunning the Allocated Memory in a Call Stack

- Next consider the following program in C:

```
// buffover.c

int main() {
    foo();
}
int foo(){
    char buffer[5]; char ch; int i = 0;
    printf("Say something: ");
    while ((ch = getchar()) != '\n')  buffer[i++] = ch;
    buffer[i] = '\0';
    printf("You said: %s\n", buffer);
    return 0;
}
```

  This program asks a user to enter a message. Whatever the user enters in a single line is accepted as the message and stored in the array **buffer** of chars. (**As the user enters keystrokes, the corresponding characters are entered into the operating system's keyboard buffer and then, when the user hits the "Enter" key on the keyboard, the operating system transfers the contents of the keyboard buffer into the `stdin` stream's internal buffer. The call to `getchar()` reads one character at a time from this buffer.**)

- Let's now see what the call stack would look like during the execution of the while loop in the program:

```
stack_ptr-->   i                  (four bytes of memory)
               ch                 (one byte of memory)
               buffer             (five bytes of memory)
               return address to top of calling stack frame

               main
```

- As the **while** loop is entering characters in the memory allocated to the array variable **buffer** on the stack, **there is no mechanism in place for stopping when the five bytes allocated to buffer are used up.**

- What happens next depends entirely on the details of how the stacks are implemented in a particular system and how the memory is allocated. If the system has the notion of a *memory word* consisting of, say, 32 bits and if stack memory is allocated at word boundaries, then as you overrun the buffer in the above program, the program will continue to function up to a point as you enter longer and longer messages in response to the prompt.

- But at some point, the string you enter will begin to overwrite the memory locations allocated to other variables on the stack and also possibly the location where the return address of the calling function is stored.

Demonstration of Program Misbehavior Because of Buffer Overflow

- I will now give a vivid demonstration of how a program may continue to function but produce incorrect results because of buffer overflow on the stack, **assuming that the overflow is not to the extent so as to cause a memory segmentation fault**.

- Let's consider the following variation on the previous program:

```
// buffover2.c

int main() {
    while(1) foo();
}

int foo(){
    unsigned int yy = 0;
    char buffer[5]; char ch; int i = 0;
    printf("Say something: ");
    while ((ch = getchar()) != '\n')  buffer[i++] = ch;
    buffer[i] = '\0';
    printf("You said: %s\n", buffer);
    printf("The variable yy: %d\n", yy);
    return 0;
}
```

- The important difference here from the previous program on Slide 17 is that we have defined a new variable **yy BEFORE** allocat-

ing memory for the array variable `buffer`. The other change here, placing the call to `foo()` inside an infinite loop in `main` is just for convenience. Now you can experiment with longer and longer input strings until you get a segfault and the program crashes.

- The call stack will now look like:

```
stack_ptr-->   i                   (four bytes of memory)
               ch                  (one byte of memory)
               buffer              (five bytes of memory)
               yy                  (four bytes)
               return address to top of calling stack frame

               main
```

Now as you enter longer and longer messages in response to the "`Say something:`" prompt, what gets written into the array `buffer` could at some point overwrite the memory allocated to the variable `yy`.

- So, whereas the program logic dictates that the value of the local variable yy should always be 0, what you actually see may depend on what string you entered in response to the prompt. When I interact with the program on my Linux laptop, I see the following behavior:

```
Say something: 012345678901234567890123467
You said: 012345678901234567890123467
```

```
The variable yy: 0                              <-----  correct

Say something: 012345678901234567890123456780
You said: 012345678901234567890123456780
The variable yy: 56                             <------  ERROR

Say something: 0123456789012345678901234567890
You said: 0123456789012345678901234567890
The variable yy: 14648                          <------  ERROR

Say something: 01234567890123456789012345678901
You said: 01234567890123456789012345678901
The variable yy: 3160376                        <------  ERROR

Say something: 012345678901234567890123456780901
You said: 012345678901234567890123456780901
The variable yy: 825243960                      <------  ERROR

....
....
```

• As you would expect, as you continue to enter longer and longer
  strings, at some point the program will crash with a segfault.

What Other Damage Can Buffer Overflow Cause

- As you have already seen, at the least you could run into unpredictable behavior from a program vulnerable to buffer overflow. **(Just imagine a banking application software written a long time ago when the great dangers of buffer overflow had not yet seeped into the collective consciousness of the programming community. A program misbehavior of the sort demonstrated on the previous slide could either make you instantly wealthy or instantly poor.)**

- From the examples shown, it should be also be clear when a buffer is overrun significantly, it would be possible to change the content of the location on the stack where the return address is stored. As mentioned previously, the return address points to the top of the stack frame of the calling function.

- In most cases, such overwrite of the return address in a stack frame would create a pointer to some invalid location in the memory. When that happens, the program will crash with a segfault.

- However, a crafty attacker may be able to cause the changed return address in a stack frame to point to another location in the

overwritten area that contains malicious executable code. This could then create serious repercussions for the system security.

- With regard to the vulnerability of the telnet service to buffer overflow attack, the problem arises when the telnet service is called with certain illegal optional flags. Since the executable for the affected telnet servers does not place any constraints on the command string it receives from a client, a client can construct a security exploit to compromise the server.