Machine-Level Programming V: Advanced Topics

15-213 / 18-213: Introduction to Computer Systems
9th Lecture, Sep. 27, 2011

Instructors:
Dave O’Hallaron, Greg Ganger, and Greg Ganger
Today

- Structures
  - Alignment
- Unions
- Memory Layout
- Buffer Overflow
  - Vulnerability
  - Protection
Structures & Alignment

- **Unaligned Data**
  
<table>
<thead>
<tr>
<th></th>
<th>i[0]</th>
<th>i[1]</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>p</td>
<td>p+1</td>
<td>p+5</td>
</tr>
<tr>
<td>i[0]</td>
<td>p+9</td>
<td>p+17</td>
<td></td>
</tr>
</tbody>
</table>

- **Aligned Data**
  
  - Primitive data type requires K bytes
  - Address must be multiple of K

```c
struct S1 {
    char c;
    int i[2];
    double v;
} *p;
```
Alignment Principles

- **Aligned Data**
  - Primitive data type requires K bytes
  - Address must be multiple of K
  - Required on some machines; advised on IA32
    - treated differently by IA32 Linux, x86-64 Linux, and Windows!

- **Motivation for Aligning Data**
  - Memory accessed by (aligned) chunks of 4 or 8 bytes (system dependent)
    - Inefficient to load or store datum that spans quad word boundaries
    - Virtual memory very tricky when datum spans 2 pages

- **Compiler**
  - Inserts gaps in structure to ensure correct alignment of fields
Specific Cases of Alignment (IA32)

- **1 byte:** `char`, ...
  - no restrictions on address

- **2 bytes:** `short`, ...
  - lowest 1 bit of address must be 0

- **4 bytes:** `int`, `float`, `char *`, ...
  - lowest 2 bits of address must be 00

- **8 bytes:** `double`, ...
  - Windows (and most other OS’s & instruction sets):
    - lowest 3 bits of address must be 000
  - Linux:
    - lowest 2 bits of address must be 00
    - i.e., treated the same as a 4-byte primitive data type

- **12 bytes:** `long double`
  - Windows, Linux:
    - lowest 2 bits of address must be 00
    - i.e., treated the same as a 4-byte primitive data type
Specific Cases of Alignment (x86-64)

- **1 byte:** char, ...
  - no restrictions on address

- **2 bytes:** short, ...
  - lowest 1 bit of address must be 0₂

- **4 bytes:** int, float, ...
  - lowest 2 bits of address must be 00₂

- **8 bytes:** double, char *, ...
  - Windows & Linux:
    - lowest 3 bits of address must be 000₂
  - Linux:
    - lowest 3 bits of address must be 000₂
    - i.e., treated the same as a 8-byte primitive data type
Satisfying Alignment with Structures

- **Within structure:**
  - Must satisfy each element’s alignment requirement

- **Overall structure placement**
  - Each structure has alignment requirement $K$
    - $K =$ Largest alignment of any element
    - Initial address & structure length must be multiples of $K$

- **Example (under Windows or x86-64):**
  - $K = 8$, due to `double` element

```c
struct S1 {
    char c;
    int i[2];
    double v;
} *p;
```
Different Alignment Conventions

- **x86-64 or IA32 Windows:**
  - \( K = 8 \), due to \texttt{double} element

```c
struct S1 {
    char c;
    int i[2];
    double v;
} *p;
```

- **IA32 Linux**
  - \( K = 4 \); \texttt{double} treated like a 4-byte data type
Meeting Overall Alignment Requirement

- For largest alignment requirement $K$
- Overall structure must be multiple of $K$

```
struct S2 {
    double v;
    int i[2];
    char c;
} *p;
```
Arrays of Structures

- Overall structure length multiple of K
- Satisfy alignment requirement for every element

```c
struct S2 {
    double v;
    int i[2];
    char c;
} a[10];
```
Accessing Array Elements

- Compute array offset 12\(i\)
  - \(\text{sizeof}(S3)\), including alignment spacers

- Element \(j\) is at offset 8 within structure

- Assembler gives offset \(a+8\)
  - Resolved during linking

```c
struct S3 {
    short i;
    float v;
    short j;
} a[10];
```

```c
short get_j(int idx) {
    return a[idx].j;
}
```

```asm
# %eax = idx
leal (%eax,%eax,2),%eax # 3*idx
movswl a+8(,%eax,4),%eax
```
Saving Space

- Put large data types first

```c
struct S4 {
  char c;
  int i;
  char d;
} *p;
```

- Effect (K=4)

```c
struct S5 {
  int i;
  char c;
  char d;
} *p;
```
Today

- **Structures**
  - Alignment
- **Unions**
- **Memory Layout**
- **Buffer Overflow**
  - Vulnerability
  - Protection
Union Allocation

- Allocate according to largest element
- Can only use one field at a time

```c
union U1 {
    char c;
    int i[2];
    double v;
} *up;

struct S1 {
    char c;
    int i[2];
    double v;
} *sp;
```
Using Union to Access Bit Patterns

typedef union {
   float f;
   unsigned u;
} bit_float_t;

float bit2float(unsigned u) {
   bit_float_t arg;
   arg.u = u;
   return arg.f;
}

unsigned float2bit(float f) {
   bit_float_t arg;
   arg.f = f;
   return arg.u;
}

Same as (float) u?

Same as (unsigned) f?
Byte Ordering Revisited

**Idea**
- Short/long/quad words stored in memory as 2/4/8 consecutive bytes
- Which is most (least) significant?
- Can cause problems when exchanging binary data between machines

**Big Endian**
- Most significant byte has lowest address
- Sparc

**Little Endian**
- Least significant byte has lowest address
- Intel x86
**Byte Ordering Example**

```c
union {
    unsigned char c[8];
    unsigned short s[4];
    unsigned int i[2];
    unsigned long l[1];
} dw;
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>s[0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s[1]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s[2]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s[3]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i[0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i[1]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>l[0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>s[0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s[1]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s[2]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s[3]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i[0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i[1]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>l[0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Byte Ordering Example (Cont).

```c
int j;
for (j = 0; j < 8; j++)
    dw.c[j] = 0xf0 + j;

printf("Characters 0-7 == \[0x%x,0x%x,0x%x,0x%x,
0x%x,0x%x,0x%x,0x%x]\n",
    dw.c[0], dw.c[1], dw.c[2], dw.c[3],
    dw.c[4], dw.c[5], dw.c[6], dw.c[7]);

printf("Shorts 0-3 == \[0x%x,0x%x,0x%x,0x%x]\n",
    dw.s[0], dw.s[1], dw.s[2], dw.s[3]);

printf("Ints 0-1 == \[0x%x,0x%x]\n",
    dw.i[0], dw.i[1]);

printf("Long 0 == \[0x%lx]\n",
    dw.l[0]);
```
Byte Ordering on IA32

Little Endian

<table>
<thead>
<tr>
<th>f0</th>
<th>f1</th>
<th>f2</th>
<th>f3</th>
<th>f4</th>
<th>f5</th>
<th>f6</th>
<th>f7</th>
</tr>
</thead>
<tbody>
<tr>
<td>i[0]</td>
<td>i[1]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>l[0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Output:

Characters 0-7 == [0xf0, 0xf1, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6, 0xf7]
Shorts 0-3 == [0xf1f0, 0xf3f2, 0xf5f4, 0xf7f6]
Ints 0-1 == [0xf3f2f1f0, 0xf7f6f5f4]
Long 0 == [0xf3f2f1f0]
## Byte Ordering on Sun

### Big Endian

<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>f0</td>
<td>f1</td>
<td>f2</td>
<td>f3</td>
</tr>
<tr>
<td>i[0]</td>
<td></td>
<td>i[1]</td>
<td></td>
</tr>
<tr>
<td>l[0]</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Output on Sun:

- **Characters**: 0–7 == [0xf0, 0xf1, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6, 0xf7]
- **Shorts**: 0–3 == [0xf0f1, 0xf2f3, 0xf4f5, 0xf6f7]
- **Ints**: 0–1 == [0xf0f1f2f3, 0xf4f5f6f7]
- **Long**: 0 == [0xf0f1f2f3]
**Byte Ordering on x86-64**

**Little Endian**

<table>
<thead>
<tr>
<th></th>
<th>f0</th>
<th>f1</th>
<th>f2</th>
<th>f3</th>
<th>f4</th>
<th>f5</th>
<th>f6</th>
<th>f7</th>
</tr>
</thead>
<tbody>
<tr>
<td>s[0]</td>
<td>s[1]</td>
<td>s[2]</td>
<td>i[0]</td>
<td>i[1]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i[0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>l[0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Output on x86-64:**

- **Characters** 0-7 == [0xf0, 0xf1, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6, 0xf7]
- **Shorts** 0-3 == [0xf1f0, 0xf3f2, 0xf5f4, 0xf7f6]
- **Ints** 0-1 == [0xf3f2f1f0, 0xf7f6f5f4]
- **Long** 0 == [0xf7f6f5f4f3f2f1f0]
Summary

- **Arrays in C**
  - Contiguous allocation of memory
  - Aligned to satisfy every element’s alignment requirement
  - Pointer to first element
  - No bounds checking

- **Structures**
  - Allocate bytes in order declared
  - Pad in middle and at end to satisfy alignment

- **Unions**
  - Overlay declarations
  - Way to circumvent type system
Today

- **Structures**
  - Alignment
- **Unions**
- **Memory Layout**
- **Buffer Overflow**
  - Vulnerability
  - Protection
IA32 Linux Memory Layout

- **Stack**
  - Runtime stack (8MB limit)
  - E.g., local variables

- **Heap**
  - Dynamically allocated storage
  - When call malloc(), calloc(), new()

- **Data**
  - Statically allocated data
  - E.g., arrays & strings declared in code

- **Text**
  - Executable machine instructions
  - Read-only

Upper 2 hex digits = 8 bits of address
Memory Allocation Example

```c
char big_array[1<<24]; /* 16 MB */
char huge_array[1<<28]; /* 256 MB */

int beyond;
char *p1, *p2, *p3, *p4;

int useless() { return 0; }

int main()
{
    p1 = malloc(1 <<28); /* 256 MB */
p2 = malloc(1 << 8); /* 256 B */
p3 = malloc(1 <<28); /* 256 MB */
p4 = malloc(1 << 8); /* 256 B */
/* Some print statements ... */
}
```

Where does everything go?
IA32 Example Addresses

address range $ \sim 2^{32}$

- $\texttt{esp}$: 0xffffbcd0
- $\texttt{p3}$: 0x65586008
- $\texttt{p1}$: 0x55585008
- $\texttt{p4}$: 0x1904a110
- $\texttt{p2}$: 0x1904a008
- $\texttt{\&p2}$: 0x18049760
- $\texttt{\&beyond}$: 0x08049744
- $\texttt{big\_array}$: 0x18049780
- $\texttt{huge\_array}$: 0x08049760
- $\texttt{main()}$: 0x080483c6
- $\texttt{useless()}$: 0x08049744
- $\texttt{final\_malloc()}$: 0x006be166

$\texttt{malloc()}$ is dynamically linked
address determined at runtime

not drawn to scale
# x86-64 Example Addresses

**address range** $\sim 2^{47}$

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>$$rsp</td>
<td>0x00007fffffffff8d1f8</td>
</tr>
<tr>
<td>p3</td>
<td>0x00002aaabaadd010</td>
</tr>
<tr>
<td>p1</td>
<td>0x00002aaaaaad010</td>
</tr>
<tr>
<td>p4</td>
<td>0x00000000011501120</td>
</tr>
<tr>
<td>p2</td>
<td>0x00000000011501010</td>
</tr>
<tr>
<td>&amp;p2</td>
<td>0x00000000010500a60</td>
</tr>
<tr>
<td>&amp;beyond</td>
<td>0x0000000000500a44</td>
</tr>
<tr>
<td>big_array</td>
<td>0x00000000010500a80</td>
</tr>
<tr>
<td>huge_array</td>
<td>0x000000000500a50</td>
</tr>
<tr>
<td>main()</td>
<td>0x0000000000400510</td>
</tr>
<tr>
<td>useless()</td>
<td>0x000000000400500</td>
</tr>
<tr>
<td>final malloc()</td>
<td>0x000000386ae6a170</td>
</tr>
</tbody>
</table>

`malloc()` is dynamically linked address determined at runtime

*not drawn to scale*
Today

- **Structures**
  - Alignment

- **Unions**

- **Memory Layout**

- **Buffer Overflow**
  - Vulnerability
  - Protection
Internet Worm and IM War

- **November, 1988**
  - Internet Worm attacks thousands of Internet hosts.
  - How did it happen?
Internet Worm and IM War

- **November, 1988**
  - Internet Worm attacks thousands of Internet hosts.
  - How did it happen?

- **July, 1999**
  - Microsoft launches MSN Messenger (instant messaging system).
  - Messenger clients can access popular AOL Instant Messaging Service (AIM) servers
Internet Worm and IM War (cont.)

- **August 1999**
  - Mysteriously, Messenger clients can no longer access AIM servers.
  - Microsoft and AOL begin the IM war:
    - AOL changes server to disallow Messenger clients
    - Microsoft makes changes to clients to defeat AOL changes.
    - At least 13 such skirmishes.
  - How did it happen?

- The Internet Worm and AOL/Microsoft War were both based on *stack buffer overflow* exploits!
  - many library functions do not check argument sizes.
  - allows target buffers to overflow.
String Library Code

■ Implementation of Unix function \texttt{gets()}

\begin{verbatim}
#include <stdio.h>

char *gets(char *dest)
{
    int c = getchar();
    char *p = dest;
    while (c != EOF && c != '\n') {
        *p++ = c;
        c = getchar();
    }
    *p = '\0';
    return dest;
}
\end{verbatim}

- No way to specify limit on number of characters to read

■ Similar problems with other library functions
  - \texttt{strncpy, strcat}: Copy strings of arbitrary length
  - \texttt{scanf, fscanf, sscanf}, when given \%s conversion specification
Vulnerable Buffer Code

```c
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}

void call_echo()
{
    echo();
}
```

```
unix>./bufdemo
Type a string: 1234567
1234567

unix>./bufdemo
Type a string: 12345678
Segmentation Fault

unix>./bufdemo
Type a string: 123456789ABC
Segmentation Fault
```
Buffer Overflow Disassembly

echo:

```
80485c5: 55          push %ebp
80485c6: 89 e5       mov %esp,%ebp
80485c8: 53          push %ebx
80485c9: 83 ec 14    sub $0x14,%esp
80485cc: 8d 5d f8    lea 0xffffffff8(%ebp),%ebx
80485cf: 89 1c 24    mov %ebx,(%esp)
80485d2: e8 9e ff ff ff call 8048575 <gets>
80485d7: 89 1c 24    mov %ebx,(%esp)
80485da: e8 05 fe ff ff ff call 80483e4 <puts@plt>
80485df: 83 c4 14    add $0x14,%esp
80485e2: 5b          pop %ebx
80485e3: 5d          pop %ebp
80485e4: c3          ret
```

call_echo:

```
80485eb: e8 d5 ff ff ff ff call 80485c5 <echo>
80485f0: c9          leave
80485f1: c3          ret
```
Buffer Overflow Stack

Before call to gets

Stack Frame for **main**

Return Address

Saved %ebp

Saved %ebx

[3] [2] [1] [0]

Stack Frame for **echo**

/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}

echo:

    pushl %ebp  # Save %ebp on stack
    movl %esp, %ebp
    pushl %ebx  # Save %ebx
    subl $20, %esp  # Allocate stack space
    leal -8(%ebp),%ebx  # Compute buf as %ebp-8
    movl %ebx, (%esp)  # Push buf on stack
    call gets  # Call gets
    ...

Before call to gets

%ebp

buf

Stack Frame for main

Return Address

Saved %ebp

Saved %ebx

[3] [2] [1] [0]

Stack Frame for echo

/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
Buffer Overflow
Stack Example

Unix>
gdb bufdemo
(gdb) break echo
Breakpoint 1 at 0x80485c9
(gdb) run
Breakpoint 1, 0x80485c9 in echo ()
(gdb) print /x $ebp
$1 = 0xffffffff88
(gdb) print /x *(unsigned *)$ebp
$2 = 0xffffffff88
(gdb) print /x *((unsigned *)$ebp + 1)
$3 = 0x80485f0

Before call to gets
Stack Frame for main

Return Address
Saved %ebp
Saved %ebx
[3][2][1][0]

Stack Frame for echo

buf

Before call to gets
Stack Frame for main

0xffffffff88

0xfffffffff

Stack Frame for echo

buf

80485eb: e8 d5 ff ff ff call 80485c5 <echo>
80485f0: c9 leave
Buffer Overflow Example #1

Before call to gets

Stack Frame for main

<table>
<thead>
<tr>
<th>08</th>
<th>04</th>
<th>85</th>
<th>f0</th>
</tr>
</thead>
<tbody>
<tr>
<td>ff</td>
<td>ff</td>
<td>d6</td>
<td>88</td>
</tr>
</tbody>
</table>

Saved %ebx

| xx | xx | xx | xx |

Stack Frame for echo

Input 1234567

Stack Frame for main

<table>
<thead>
<tr>
<th>08</th>
<th>04</th>
<th>85</th>
<th>f0</th>
</tr>
</thead>
<tbody>
<tr>
<td>ff</td>
<td>ff</td>
<td>d6</td>
<td>88</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>00</th>
<th>37</th>
<th>36</th>
<th>35</th>
</tr>
</thead>
<tbody>
<tr>
<td>34</td>
<td>33</td>
<td>32</td>
<td>31</td>
</tr>
</tbody>
</table>

Stack Frame for echo

Overflow buf, and corrupt %ebx, but no problem
Buffer Overflow Example #2

Before call to gets

Stack Frame for main

0x00000000 0x00000004 0x00000085 0x000000f0
ff ff d6 88
Saved %ebx
xx xx xx xx

buf

Stack Frame for echo

0xfffffd678 0xfffffd688

Input 12345678

Stack Frame for main

0x00000000 0x00000004 0x00000085 0x000000f0
ff ff d6 00

buf

Base pointer corrupted

80485eb: e8 d5 ff ff ff call 80485c5 <echo>
80485f0: c9 leave # Set %ebp to corrupted value
80485f1: c3 ret
Buffer Overflow Example #3

Before call to `gets`

Stack Frame for `main`

```
08 04 85 f0
ff ff d6 88
```

Saved `%ebx`
```
xx xx xx xx
```

Stack Frame for `echo`
```
0xfffffd678
```

Input 123456789ABC

Stack Frame for `main`
```
08 04 85 00
43 42 41 39
38 37 36 35
34 33 32 31
```

Stack Frame for `echo`
```
0xfffffd688
```

Return address corrupted

```
80485eb:  e8 d5 ff ff ff  call  80485c5 <echo>
80485f0:  c9          leave  # Desired return point
```
Malicious Use of Buffer Overflow

- Input string contains byte representation of executable code
- Overwrite return address A with address of buffer B
- When `bar()` executes `ret`, will jump to exploit code
Exploits Based on Buffer Overflows

- Buffer overflow bugs allow remote machines to execute arbitrary code on victim machines

- Internet worm
  - Early versions of the finger server (fingerd) used `gets()` to read the argument sent by the client:
    - `finger droh@cs.cmu.edu`
  - Worm attacked fingerd server by sending phony argument:
    - `finger "exploit-code padding new-return-address"
  - exploit code: executed a root shell on the victim machine with a direct TCP connection to the attacker.
Exploits Based on Buffer Overflows

- **Buffer overflow bugs allow remote machines to execute arbitrary code on victim machines**

- **IM War**
  - AOL exploited existing buffer overflow bug in AIM clients
  - exploit code: returned 4-byte signature (the bytes at some location in the AIM client) to server.
  - When Microsoft changed code to match signature, AOL changed signature location.
Date: Wed, 11 Aug 1999 11:30:57 -0700 (PDT)
From: Phil Bucking <philbucking@yahoo.com>
Subject: AOL exploiting buffer overrun bug in their own software!
To: rms@pharlap.com

Mr. Smith,

I am writing you because I have discovered something that I think you might find interesting because you are an Internet security expert with experience in this area. I have also tried to contact AOL but received no response.

I am a developer who has been working on a revolutionary new instant messaging client that should be released later this year.

... It appears that the AIM client has a buffer overrun bug. By itself this might not be the end of the world, as MS surely has had its share. But AOL is now *exploiting their own buffer overrun bug* to help in its efforts to block MS Instant Messenger.

Since you have significant credibility with the press I hope that you can use this information to help inform people that behind AOL's friendly exterior they are nefariously compromising peoples' security.

Sincerely,
Phil Bucking
Founder, Bucking Consulting
philbucking@yahoo.com

It was later determined that this email originated from within Microsoft!
Avoiding Overflow Vulnerability

/* Echo Line */
void echo()
{
    char buf[4]; /* Way too small! */
    fgets(buf, 4, stdin);
    puts(buf);
}

Use library routines that limit string lengths
  - `fgets` instead of `gets`
  - `strncpy` instead of `strcpy`
  - Don’t use `scanf` with `%s` conversion specification
    - Use `fgets` to read the string
    - Or use `%ns` where `n` is a suitable integer
System-Level Protections

- Randomized stack offsets
  - At start of program, allocate random amount of space on stack
  - Makes it difficult for hacker to predict beginning of inserted code

- Nonexecutable code segments
  - In traditional x86, can mark region of memory as either “read-only” or “writeable”
    - Can execute anything readable
  - X86-64 added explicit “execute” permission

```
unix> gdb bufdemo
(gdb) break echo
(gdb) run
(gdb) print /x $ebp
$1 = 0xfffffc638
(gdb) run
(gdb) print /x $ebp
$2 = 0xfffffbb08
(gdb) run
(gdb) print /x $ebp
$3 = 0xfffffc6a8
```
Stack Canaries

**Idea**
- Place special value ("canary") on stack just beyond buffer
- Check for corruption before exiting function

**GCC Implementation**
- `-fstack-protector`
- `-fstack-protector-all`

```
unix> ./bufdemo-protected
Type a string: 1234
1234

unix> ./bufdemo-protected
Type a string: 12345
*** stack smashing detected ***
```
### Protected Buffer Disassembly

```
804864d:  55                 push  %ebp
804864e:  89 e5              mov   %esp,%ebp
8048650:  53                 push  %ebx
8048651:  83 ec 14           sub   $0x14,%esp
8048654:  65 a1 14 00 00 00  mov   %gs:0x14,%eax
804865a:  89 45 f8           mov   %eax,0xffffffff8(%ebp)
804865d:  31 c0              xor   %eax,%eax
804865f:  6d 5d f4           lea    0xffffffff4(%ebp),%ebx
8048662:  89 1c 24           mov   %ebx,(%esp)
8048665:  e8 77 ff ff ff     call   80485e1 <gets>
804866a:  e8 ca fd ff ff     call   804843c <puts@plt>
8048672:  8b 45 f8           mov   0xffffffff8(%ebp),%eax
8048675:  65 33 05 14 00 00 00 xor   %gs:0x14,%eax
804867c:  74 05              je     8048683 <echo+0x36>
804867e:  e8 a9 fd ff ff     call   804842c <FAIL>
8048683:  83 c4 14           add   $0x14,%esp
8048686:  5b                 pop   %ebx
8048687:  5d                 pop   %ebp
8048688:  c3                 ret
```

### echo:

```
804864d:  55                 push  %ebp
804864e:  89 e5              mov   %esp,%ebp
8048650:  53                 push  %ebx
8048651:  83 ec 14           sub   $0x14,%esp
8048654:  65 a1 14 00 00 00  mov   %gs:0x14,%eax
804865a:  89 45 f8           mov   %eax,0xffffffff8(%ebp)
804865d:  31 c0              xor   %eax,%eax
804865f:  6d 5d f4           lea    0xffffffff4(%ebp),%ebx
8048662:  89 1c 24           mov   %ebx,(%esp)
8048665:  e8 77 ff ff ff     call   80485e1 <gets>
804866a:  e8 ca fd ff ff     call   804843c <puts@plt>
8048672:  8b 45 f8           mov   0xffffffff8(%ebp),%eax
8048675:  65 33 05 14 00 00 00 xor   %gs:0x14,%eax
804867c:  74 05              je     8048683 <echo+0x36>
804867e:  e8 a9 fd ff ff     call   804842c <FAIL>
8048683:  83 c4 14           add   $0x14,%esp
8048686:  5b                 pop   %ebx
8048687:  5d                 pop   %ebp
8048688:  c3                 ret
```
Setting Up Canary

Before call to gets

Stack Frame for main

Return Address
Saved %ebp
Saved %ebx
Canary

[3][2][1][0]

Stack Frame for echo

/* Echo Line */
void echo()
{
    char buf[4]; /* Way too small! */
    gets(buf);
    puts(buf);
}

Return Address
Saved %ebp
%ebp

Stack Frame for echo

echo:
.
.
movl %gs:20, %eax    # Get canary
movl %eax, -8(%ebp)  # Put on stack
xorl %eax, %eax     # Erase canary
.
.

### Checking Canary

**Before call to gets**

<table>
<thead>
<tr>
<th>Stack Frame for <strong>main</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Return Address</td>
</tr>
<tr>
<td>Saved %ebp</td>
</tr>
<tr>
<td>Saved %ebx</td>
</tr>
<tr>
<td>Canary</td>
</tr>
<tr>
<td>[3][2][1][0]</td>
</tr>
<tr>
<td>Stack Frame for <strong>echo</strong></td>
</tr>
</tbody>
</table>

```c
/* Echo Line */
void echo()
{
    char buf[4]; /* Way too small! */
    gets(buf);
    puts(buf);
}
```

**Echo:**

```assembly
...  
movl  -8(%ebp), %eax        # Retrieve from stack
xorl  %gs:20, %eax          # Compare with Canary
je    .L24                 # Same: skip ahead
    call  __stack_chk_fail   # ERROR
  .L24:
    ...  
```
Canary Example

Before call to gets

Stack Frame for **main**

- Return Address
- Saved %ebp
- Saved %ebx
- buf

Stack Frame for **echo**

- Saved %ebp
- Saved %ebx
- Stack Frame for main

Input 1234

Stack Frame for **main**

- Return Address
- Saved %ebp
- Saved %ebx
- buf

Stack Frame for **echo**

- Saved %ebp
- Saved %ebx

Benign corruption!
(allow programmers to make silent off-by-one errors)

(gdb) break echo
(gdb) run
(gdb) stepi 3
(gdb) print /x *((unsigned *) $ebp - 2)
$1 = 0x3e37d00
Worms and Viruses

- **Worm: A program that**
  - Can run by itself
  - Can propagate a fully working version of itself to other computers

- **Virus: Code that**
  - Add itself to other programs
  - Cannot run independently

- **Both are (usually) designed to spread among computers and to wreak havoc**
Today

- Structures
  - Alignment
- Unions
- Memory Layout
- Buffer Overflow
  - Vulnerability
  - Protection