Machine-Level Programming V: Advanced Topics  
15-213/18-243, Spring 2011  
9th Lecture, Feb. 12th  
Instructors:  
Anthony Rowe, Seth Goldstein and Gregory Kesden

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

Structures & Alignment  
- Unaligned Data  
  - Primitive data type requires K bytes  
  - Address must be multiple of K  

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

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 0
- **4 bytes**: int, float, ...
  - lowest 2 bits of address must be 00
- **8 bytes**: double, char *, ...
  - Windows & Linux:
    - lowest 3 bits of address must be 000
    - i.e., treated the same as a 4-byte primitive data type
- **16 bytes**: long double
  - Linux:
    - Lowest 3 bits of address must be 000
    - i.e., treated the same as a 8-byte primitive data type

Alignment Examples

<table>
<thead>
<tr>
<th>Address (hex)</th>
<th>Address (binary)</th>
<th>Alignment</th>
<th>Types (IA32)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>0b00000000</td>
<td>1,2,4,8...</td>
<td>char, short, int, float, *, double</td>
</tr>
<tr>
<td>0x01</td>
<td>0b00000001</td>
<td>1</td>
<td>char</td>
</tr>
<tr>
<td>0x02</td>
<td>0b00000010</td>
<td>1,2</td>
<td>char, short</td>
</tr>
<tr>
<td>0x03</td>
<td>0b00000011</td>
<td>1</td>
<td>char</td>
</tr>
<tr>
<td>0x04</td>
<td>0b00000100</td>
<td>1,2,4</td>
<td>char, short, int, float, *</td>
</tr>
<tr>
<td>0x05</td>
<td>0b00000101</td>
<td>1</td>
<td>char</td>
</tr>
<tr>
<td>0x06</td>
<td>0b00000110</td>
<td>1,2</td>
<td>char, short</td>
</tr>
<tr>
<td>0x07</td>
<td>0b00000111</td>
<td>1</td>
<td>char</td>
</tr>
<tr>
<td>0x08</td>
<td>0b00001000</td>
<td>1,2,4,8</td>
<td>char, short, int, float, *, double</td>
</tr>
<tr>
<td>0x09</td>
<td>0b00001001</td>
<td>1</td>
<td>char</td>
</tr>
<tr>
<td>0x0a</td>
<td>0b00001010</td>
<td>1,2</td>
<td>char, short</td>
</tr>
<tr>
<td>0x0b</td>
<td>0b00001011</td>
<td>1</td>
<td>char</td>
</tr>
<tr>
<td>0x0c</td>
<td>0b00001100</td>
<td>1,2,4</td>
<td>char, short, int, float, *</td>
</tr>
</tbody>
</table>

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

- IA32 Linux
  - K = 4; double treated like a 4-byte data type
  ```
  struct S1 {
    char c;
    int i[2];
    double v;
  } *
p;
  ```

**Meeting Overall Alignment Requirement**

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

**Arrays of Structures**

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

**Accessing Array Elements**

- Compute array offset 12i
  - `sizeof(S3)`, including alignment spacers
- Element j is at offset 8 within structure
- Assembler gives offset a+8
  - Resolved during linking

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

c 3 bytes  i 1 byte  d 1 byte

```
i 1 byte  d 1 byte
```

Union Allocation

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

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

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

c 1 byte  i[0]  i[1]  4 bytes  v

```
up+0  up+4  up+8
```

Using Union to Access Bit Patterns

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

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

```c
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;
```

32-bit

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</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>

64-bit

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</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>

### 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%x]\n",
      dw.l[0]);
```

### Byte Ordering on IA32

**Little Endian**

```
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]  
",
      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]  
",
      dw.i[0], dw.i[1]);
printf("Long 0 == [0x%x]\n",
      dw.l[0]);
```

Output:

- **Characters 0-7**
  - Little Endian: [0xf0,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7]
- **Shorts 0-3**
  - Little Endian: [0x1f00,0x1f01,0x1f02,0x1f03]
- **Ints 0-1**
  - Little Endian: [0x1f0,0x1f1]
- **Long 0**
  - Little Endian: [0x1f0]

5
**Byte Ordering on Sun**

**Big Endian**

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</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 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>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</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 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

---

**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 ~2\(^32\)*

<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$esp</td>
<td>0xffffbcd0</td>
</tr>
<tr>
<td>p3</td>
<td>0x65586008</td>
</tr>
<tr>
<td>p1</td>
<td>0x55585008</td>
</tr>
<tr>
<td>p4</td>
<td>0x1904a110</td>
</tr>
<tr>
<td>p2</td>
<td>0x1904a008</td>
</tr>
<tr>
<td>&amp;p2</td>
<td>0x18049760</td>
</tr>
<tr>
<td>&amp;beyond</td>
<td>0x18049744</td>
</tr>
<tr>
<td>big_array</td>
<td>0x18049780</td>
</tr>
<tr>
<td>huge_array</td>
<td>0x08049760</td>
</tr>
<tr>
<td>main()</td>
<td>0x080483c6</td>
</tr>
<tr>
<td>useless()</td>
<td>0x08049744</td>
</tr>
<tr>
<td>final malloc()</td>
<td>0x006be166</td>
</tr>
</tbody>
</table>

malloc() is dynamically linked
address determined at runtime

---

**x86-64 Example Addresses**

*address range ~2\(^{47}\)*

<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$rsp</td>
<td>0x000007fffff8d1f8</td>
</tr>
<tr>
<td>p3</td>
<td>0x000002aaaaad0d10</td>
</tr>
<tr>
<td>p1</td>
<td>0x000002aaaaada0c10</td>
</tr>
<tr>
<td>p4</td>
<td>0x0000000011501120</td>
</tr>
<tr>
<td>p2</td>
<td>0x0000000011501010</td>
</tr>
<tr>
<td>&amp;p2</td>
<td>0x0000000001050a60</td>
</tr>
<tr>
<td>&amp;beyond</td>
<td>0x000000000050a44</td>
</tr>
<tr>
<td>big_array</td>
<td>0x000000001050a80</td>
</tr>
<tr>
<td>huge_array</td>
<td>0x00000000050a50</td>
</tr>
<tr>
<td>main()</td>
<td>0x000000000400f30</td>
</tr>
<tr>
<td>useless()</td>
<td>0x000000000040f30</td>
</tr>
<tr>
<td>final malloc()</td>
<td>0x00000000386ae6a170</td>
</tr>
</tbody>
</table>

malloc() is dynamically linked
address determined at runtime
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?

- 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 `gets()`

```c
/* Get string from stdin */
char *gets(char *dest)
{
    int c = getchar();
    char *p = dest;
    while (c != EOF && c != '\n') {
        *p++ = c;
        c = getchar();
    }
    *p = '\0';
    return dest;
}
```

- No way to specify limit on number of characters to read
- Similar problems with other library functions
  - `strcpy`, `strcat`: Copy strings of arbitrary length
  - `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);
}
```

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

Before call to `gets`

<table>
<thead>
<tr>
<th>Saved %ebp</th>
<th>Saved %ebx</th>
<th>Saved %esp</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0]</td>
<td>[1]</td>
<td>[2]</td>
</tr>
</tbody>
</table>

`buf`: `0x100000010`

After call to `gets`

<table>
<thead>
<tr>
<th>Saved %ebp</th>
<th>Saved %ebx</th>
<th>Saved %esp</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0]</td>
<td>[1]</td>
<td>[2]</td>
</tr>
</tbody>
</table>

Echo:

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

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

Before call to `gets`

<table>
<thead>
<tr>
<th>Saved %ebp</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0]</td>
</tr>
</tbody>
</table>

`buf`: `0x100000010`

After call to `gets`

<table>
<thead>
<tr>
<th>Saved %ebp</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0]</td>
</tr>
</tbody>
</table>

Example stack frames for `main` and `echo` functions.
Buffer Overflow Example

### Example #1

Before call to `gets`
```
Stack Frame for main
```
```
Stack Frame for echo
```
```
Base pointer corrupted
```
Input 12345678
```
```
Return address corrupted
```

### Example #2

Before call to `gets`
```
Stack Frame for main
```
```
Stack Frame for echo
```
```
Before call to `gets`
```
```
Buffer Overflow Example #3

Before call to `gets`
```
Stack Frame for main
```
```
Stack Frame for echo
```
```
Buffer Overflow Example #3

Before call to `gets`
```
Stack Frame for main
```
```
Stack Frame for echo
```
```
Before call to `gets`
```
```
Base pointer corrupted
```
Input 12345678
```
```
Return address corrupted
```

### Example #3

Before call to `gets`
```
Stack Frame for main
```
```
Stack Frame for echo
```
```
Before call to `gets`
```
```
Buffer Overflow Example #3

Before call to `gets`
```
Stack Frame for main
```
```
Stack Frame for echo
```
```
Before call to `gets`
```
```
Base pointer corrupted
```
Input 12345678
```
```
Return address corrupted
```

### Example #2

Before call to `gets`
```
Stack Frame for main
```
```
Stack Frame for echo
```
```
Before call to `gets`
```
```
Base pointer corrupted
```
Input 12345678
```
```
Return address corrupted
```

### Example #3

Before call to `gets`
```
Stack Frame for main
```
```
Stack Frame for echo
```
```
Before call to `gets`
```
```
Base pointer corrupted
```
Input 12345678
```
```
Return address corrupted
```
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**
- **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.

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 drzh@cs.cmu.edu`
  - Worm attacked finger 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.

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@phaslap.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!

- MS IIS webserver vulnerability
- Starts 100 threads running
- Spread self
  - Generate random IP addresses & send attack string
  - Between 1st & 19th of month
- Attack www.whitehouse.gov
  - Send 98,304 packets; sleep for 4-1/2 hours; repeat
    - Denial of service attack
    - Between 21st & 27th of month
- Deface server's home page
  - After waiting 2 hours

Avoiding Overflow Vulnerability

/* Echo Line */
#define MAX_STR_LEN 4

void echo()
{
    char buf[MAX_STR_LEN]; /* Way too small! */
    fgets(buf, MAX_STR_LEN, 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
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`

```bash
unix>./bufdemo-protected
Type a string: 1234
1234
unix>./bufdemo-protected
*** stack smashing detected ***
```

Protected Buffer Disassembly

```assembly
804866d: 55  push %ebp
804866e: 89 e5  mov %esp,%ebp
804866f: 83 ec 14  sub $0x14,%esp
8048670: 65 al 14 00 00 00  mov $%gs:0x14,%eax
8048675: 89 45 f8  mov %eax,%xmm
8048676: 53  push %ebx
8048677: 8d 5d f4  lea 0xfffffff4(%ebp),%ebx
8048678: 89 1c 24  mov %ebx,(%esp)
804867b: e8 ca fd ff ff  call 804842c <FAIL>
804867e: 74 05  je     8048683 <echo+0x36>
8048680: c3  ret
```

Setting Up Canary

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

Checking Canary

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

Before call to `gets`

<table>
<thead>
<tr>
<th>Stack Frame for <code>main</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>Return Address</td>
</tr>
<tr>
<td>Saved <code>%ebp</code></td>
</tr>
<tr>
<td>Saved <code>%ebx</code></td>
</tr>
<tr>
<td>buf</td>
</tr>
<tr>
<td>Stack Frame for <code>echo</code></td>
</tr>
</tbody>
</table>

Input `1234`

<table>
<thead>
<tr>
<th>Stack Frame for <code>main</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>Return Address</td>
</tr>
<tr>
<td>Saved <code>%ebp</code></td>
</tr>
<tr>
<td>Saved <code>%ebx</code></td>
</tr>
<tr>
<td>buf</td>
</tr>
<tr>
<td>Stack Frame for <code>echo</code></td>
</tr>
</tbody>
</table>

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

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