Machine-Level Programming V: Advanced Topics

15-213 / 18-213: Introduction to Computer Systems
9th Lecture, Sep. 24, 2013

Instructors:
Randy Bryant, Dave O’Hallaron, and Greg Kesden
Today

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

- **Unaligned Data**

  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

```
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\(_2\)

- **4 bytes: int, float, char *, ...**
  - lowest 2 bits of address must be 00\(_2\)

- **8 bytes: double, ...**
  - Windows (and most other OS’s & instruction sets):
    - lowest 3 bits of address must be 000\(_2\)
  - Linux:
    - lowest 2 bits of address must be 00\(_2\)
    - i.e., treated the same as a 4-byte primitive data type

- **12 bytes: long double**
  - Windows (GCC), Linux:
    - lowest 2 bits of address must be 00\(_2\)
    - 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, long, char *, ...**
  - lowest 3 bits of address must be 000₂

- **16 bytes: long double (GCC on Linux or Windows)**
  - lowest 4 bits of address must be 0000₂
Satisfying Alignment with Structures

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

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

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

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

- Windows, x86-64
  - $K = 8$, due to `double` element

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

- IA32 Linux
  - $K = 4$; `double` treated like a 4-byte data type
Meeting Overall Alignment Requirement (Windows, x86-64)

- 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 (Windows, x86-64)

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

```c
struct S2 {
    double v;
    int i[2];
    char c;
} a[10];
```
Meeting Overall Alignment Requirement (IA32 Linux)

- For largest alignment requirement $K$
- Overall structure must be multiple of $K$
  - Up to maximum of $K=4$

```
struct S2 {
    double v;
    int i[2];
    char c;
} *p;
```
Arrays of Structures (IA32 Linux)

- 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*idx**
  - `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;
}
```

```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

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

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

(Windows or x86-64)
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 byte 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, ARM Android and IOS

- **Bi Endian**
  - Can be configured either way
  - ARM
Byte Ordering Example

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>i[0]</td>
<td></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>
<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>i[0]</td>
<td></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>
<td></td>
</tr>
</tbody>
</table>
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

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>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 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]

Print
Byte Ordering on x86-64

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>
</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
    - IA32 Linux unusual in only requiring 4-byte alignment for 8-byte data
  - 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., global vars, static vars, strings

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

Upper 2 hex digits = 8 bits of address
not drawn to scale
Memory Allocation Example

```
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 $2^{32}$

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

malloc() is dynamically linked
address determined at runtime
x86-64 Example Addresses

address range $\sim 2^{47}$

- $\texttt{rsp}$: 0x00007ffffff8d1f8
- $\texttt{p3}$: 0x000002aaabaadd010
- $\texttt{p1}$: 0x000002aaaaaad010
- $\texttt{p4}$: 0x0000000011501120
- $\texttt{p2}$: 0x0000000011501010
- $\&\texttt{p2}$: 0x00000000010500a60
- $\&\texttt{beyond}$: 0x00000000000500a44
- $\texttt{big_array}$: 0x0000000010500a80
- $\texttt{huge_array}$: 0x0000000000500a50
- $\texttt{main()}$: 0x00000000000400510
- $\texttt{useless()}$: 0x00000000000400500
- $\texttt{final \texttt{malloc}()}$: 0x000000386ae6a170

$\texttt{malloc()}$ 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

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

- 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 != '
') {
        *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);
}

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

```bash
unix> ./bufdemo
Type a string:0123456789a
0123456789a

unix> ./bufdemo
Type a string:0123456789ab
Segmentation Fault
```
Buffer Overflow Disassembly

echo:

```
080485c3  <echo>:
80485c3:  55                      push   %ebp
80485c4:  89 e5                   mov    %esp,%ebp
80485c6:  53                      push   %ebx
80485c7:  83 ec 24                sub    $0x24,%esp
80485ca:  8d 5d f4                lea    -0xc(%ebp),%ebx
80485cd:  89 1c 24                mov    %ebx,(%esp)
80485d0:  e8 9e ff ff ff          call   8048573 <gets>
80485d5:  89 1c 24                mov    %ebx,(%esp)
80485d8:  e8 2f fe ff ff ff      call   804840c <puts@plt>
80485dd:  83 c4 24                add    $0x24,%esp
80485e0:  5b                      pop    %ebx
80485e1:  5d                      pop    %ebp
80485e2:  c3                      ret
```

call_echo:

```
...
80485e9:  e8 d5 ff ff ff ff      call   80485c3 <echo>
80485ee:  c9                      leave
80485ef:  c3                      ret
```
Buffer Overflow Stack

### 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 %ebp</td>
</tr>
<tr>
<td>Saved %ebx</td>
</tr>
<tr>
<td>4 bytes unused</td>
</tr>
<tr>
<td>[3] [2] [1] [0]</td>
</tr>
</tbody>
</table>

```assembly
[3][2][1][0] buf
```

| Stack Frame for `echo` |

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

### Echo Line:

```c
/* Echo Line */
```

```assembly
pushl %ebp # Save %ebp on stack
movl %esp, %ebp # Save %ebx
pushl %ebx # Save %ebx
subl $36, %esp # Allocate stack space
leal -12(%ebp),%ebx # Compute buf as %ebp-12
movl %ebx, (%esp) # Push buf on stack
call gets # Call gets
...
Buffer Overflow
Stack Example

Before call to gets

Stack Frame for main

Return Address
Saved %ebp
Saved %ebx
4 bytes unused

Stack Frame for echo

Before call to gets

Stack Frame for main

Return address
%ebp

Stack Frame for echo

unix> gdb bufdemo
(gdb) break echo
Breakpoint 1 at 0x80485c9
(gdb) run
Breakpoint 1, 0x80485c9 in echo ()
(gdb) print /x $ebp
$1 = 0xfffffd248
(gdb) print /x *(unsigned *)$ebp
$2 = 0xfffffd258
(gdb) print /x *((unsigned *)$ebp + 1)
$3 = 0x80485ee
(gdb) print /x *((unsigned *)$ebp - 1)
$4 = 0x2c3ff4

Stack Frame for main

Before call to gets

Stack Frame for echo

80485e9:   e8 d5 ff ff ff
80485ee:   c9

call 80485c3 <echo>
leave
### Buffer Overflow Example #1

**Before call to gets**

<table>
<thead>
<tr>
<th>Stack Frame for main</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000000</td>
</tr>
<tr>
<td>0xff</td>
</tr>
<tr>
<td>0x00</td>
</tr>
<tr>
<td>xx</td>
</tr>
</tbody>
</table>

**Saved %ebx**

| 0x00000000 | 0x0000002c | 0x0000003f | 0x000000f4 |

**buf**

| xx | xx | xx | xx |

**Input 0123456789a**

<table>
<thead>
<tr>
<th>Stack Frame for main</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000000</td>
</tr>
<tr>
<td>0xff</td>
</tr>
<tr>
<td>0x00</td>
</tr>
<tr>
<td>0x37</td>
</tr>
<tr>
<td>0x33</td>
</tr>
</tbody>
</table>

**Saved %ebx**

| 0x00000000 | 0x00000061 | 0x00000039 | 0x00000038 |

**buf**

| 0x37 | 0x36 | 0x35 | 0x34 |

**Overflow buf, and corrupt %ebx, but no adverse effects**
Buffer Overflow Example #2

Before call to gets

Stack Frame for main

0xffffffffd258

return address 08 04 85 ee
%ebp ff ff d2 58
Saved %ebx 00 2c 3f f4
buf xx xx xx xx

Stack Frame for echo

Base pointer corrupted!

Input 0123456789ab

Stack Frame for main

return address 08 04 85 ee
%ebp ff ff d2 00
Saved %ebx 62 61 39 38
buf 37 36 35 34

Stack Frame for echo

0xffffffffd200

80485e9: e8 d5 ff ff ff
80485ee: c9
80485ef: c3

call 80485c3 <echo>
leave # Set %ebp to bad value
ret
Buffer Overflow Example #3

Before call to gets

Stack Frame for main

return address
%ebp
Saved %ebx
buf

Stack Frame for main

Input 0123456789abcdef

Stack Frame for main

Saved %ebx
buf

Return address corrupted!

0xffffffffd258
return address
%ebp
Saved %ebx
buf

0xffffffffd200

... 
80485e9: e8 d5 ff ff ff
80485ee: c9

call 80485c3 <echo>
leave # Desired return point
Buffer Overflow Example #4

- Can we trick program into calling a different function?

```c
void gotcha() {
    printf("This function should not get called!\n");
}
```

- Idea: Alter return address on stack

```
0x0804861e <gotcha>:  
0x804861e: 55 push %ebp 
   ... 
```
Buffer Overflow Example #4

- **Alter return address on stack**
  
  0x804861e <gotcha>:
  
  804861e:  55  push %ebp
  
  . . .

- **Exploit string:**
  
  00 00 00 00 00 00 00 00 (8X)
  
  00 00 00 00 00 00 00 00 (8X)
  
  1e 86 04 08 (Little Endian)

- **Must supply as raw bytes**
  
  - E.g., via tool hex2raw
  
  - See Buffer Lab
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

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
  - Currently disabled on shark machines

- 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

```bash
unix> gdb bufdemo
(gdb) break echo
(gdb) run
(gdb) print /x $ebp
$1 = 0xfffffffc638
(gdb) run
(gdb) print /x $ebp
$2 = 0xffffffbb08
(gdb) run
(gdb) print /x $ebp
$3 = 0xfffffffc6a8
```
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:123
123

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

```
804864b:  55                  push   %ebp
804864c:  89 e5                mov    %esp,%ebp
804864e:  53                   push   %ebx
804864f:  83 ec 24             sub    $0x24,%esp
8048652:  65 a1 14 00 00 00    mov    %gs:0x14,%eax
8048658:  89 45 f4             mov    %eax,-0xc(%ebp)
804865b:  31 c0                xor    %eax,%eax
804865d:  8d 5d f0             lea    -0x10(%ebp),%ebx
8048660:  89 1c 24             mov    %ebx,(%esp)
8048663:  e8 77 ff ff ff       call   80485df <gets>
8048668:  89 1c 24             mov    %ebx,(%esp)
804866b:  e8 f0 fd ff ff       call   8048460 <puts@plt>
804866e:  8b 45 f4             mov    %eax,%eax
8048670:  65 33 05 14 00 00 00 xor    %gs:0x14,%eax
8048673:  74 05                je     8048681 <echo+0x36>
8048676:  e8 cf fd ff ff       call   8048450 <...fail...>
804867b:  83 c4 24             add    $0x24,%esp
8048681:  5b                   pop    %ebx
8048684:  5d                   pop    %ebp
8048686:  c3                   ret
```

echo:

```
804864b:  55                  push   %ebp
804864c:  89 e5                mov    %esp,%ebp
804864e:  53                   push   %ebx
804864f:  83 ec 24             sub    $0x24,%esp
8048652:  65 a1 14 00 00 00    mov    %gs:0x14,%eax
8048658:  89 45 f4             mov    %eax,-0xc(%ebp)
804865b:  31 c0                xor    %eax,%eax
804865d:  8d 5d f0             lea    -0x10(%ebp),%ebx
8048660:  89 1c 24             mov    %ebx,(%esp)
8048663:  e8 77 ff ff ff       call   80485df <gets>
8048668:  89 1c 24             mov    %ebx,(%esp)
804866b:  e8 f0 fd ff ff       call   8048460 <puts@plt>
804866e:  8b 45 f4             mov    %eax,%eax
8048670:  65 33 05 14 00 00 00 xor    %gs:0x14,%eax
8048673:  74 05                je     8048681 <echo+0x36>
8048676:  e8 cf fd ff ff       call   8048450 <...fail...>
804867b:  83 c4 24             add    $0x24,%esp
8048681:  5b                   pop    %ebx
8048684:  5d                   pop    %ebp
8048686:  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);
}

Before call to gets

%ebp

buf

echo:
    .
    movl %gs:20, %eax    # Get canary
    movl %eax, -12(%ebp) # Put on stack
    xorl %eax, %eax      # Erase canary
    .

Return Address

Saved %ebx

Canary
Checking Canary

Before call to `gets`

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

```
movl  -12(%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**

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

<table>
<thead>
<tr>
<th>Stack Frame for echo</th>
</tr>
</thead>
<tbody>
<tr>
<td>da fc cb 23</td>
</tr>
<tr>
<td>[3][2][1][0]</td>
</tr>
</tbody>
</table>

**Input 1234**

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

<table>
<thead>
<tr>
<th>Stack Frame for echo</th>
</tr>
</thead>
<tbody>
<tr>
<td>da fc cb 00</td>
</tr>
<tr>
<td>34 33 32 31</td>
</tr>
</tbody>
</table>

(gdb) break echo
(gdb) run
(gdb) stepti 3
(gdb) print /x *((unsigned *) $ebp - 3)
$1 = dafcccb23

Canary corrupted
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
Summary

- **Structures**
  - Alignment

- **Unions**

- **Memory Layout**

- **Buffer Overflow**
  - Vulnerability
  - Protection