CS 33

Machine Programming (6)
Buffer Overflow Attack Revisited

```c
int main( ) {
    char buf[80];
    gets(buf);
    puts(buf);
    return 0;
}
```

main:

```
subq $88, %rsp  # grow stack
movq %rsp, %rdi  # setup arg
call gets
movq %rsp, %rdi  # setup arg
call puts
movl $0, %eax  # set return value
addq $88, %rsp  # pop stack
ret
```
Stack Randomization

- We don't know exactly where the stack is
  - buffer is 2000 bytes long
  - the location of the buffer might be anywhere between 7000 and 8000

![Diagram showing stack randomization](image)
NOP Slides

• NOP (No-Op) instructions do nothing
  – they just increment %rip to point to the next instruction
  – they are each one-byte long
  – a sequence of n NOPs occupies n bytes
    » if executed, they effectively add n to %rip
    » execution "slides" through them
NOP Slides and Stack Randomization

previous frame

9000
8000
1000-byte exploit
1000-byte NOP slide
7000

1000-byte NOP slide

previous frame

10000
8000
1000-byte exploit
1000-byte NOP slide
8000

1000-byte NOP slide
Recursive Function

```c
/* Recursive popcount */
int pcount_r(unsigned x) {
    if (x == 0)
        return 0;
    else return
        (x & 1) + pcount_r(x >> 1);
}
```

- Registers
  - `%eax`, `%edx` used without first saving
  - `%ebx` used, but saved at beginning & restored at end

```
pcount_r:
pushl %ebp
movl %esp, %ebp
pushl %ebx
subl $4, %esp
movl 8(%ebp), %ebx
movl $0, %eax
testl %ebx, %ebx
je .L3
movl %ebx, %eax
shrl $1, %eax
movl %eax, (%esp)
call pcount_r
movl %ebx, %edx
andl $1, %edx
leal (%edx,%eax), %eax
.L3:
addl $4, %esp
popl %ebx
popl %ebp
ret
```
Tail Recursion

```c
int factorial(int x) {
    if (x == 1)
        return x;
    else
        return x * factorial(x - 1);
}
```

```c
int factorial(int x) {
    return f2(x, 1);
}
```

```c
int f2(int a1, int a2) {
    if (a1 == 1)
        return a2;
    else
        return f2(a1 - 1, a1 * a2);
}
```
No Tail Recursion (1)

```
x: 6
return addr
x: 5
return addr
x: 4
return addr
x: 3
return addr
x: 2
return addr
x: 1
return addr
```
No Tail Recursion (2)

<table>
<thead>
<tr>
<th>x: 6</th>
<th>ret: 720</th>
</tr>
</thead>
<tbody>
<tr>
<td>return addr</td>
<td></td>
</tr>
<tr>
<td>x: 5</td>
<td>ret: 120</td>
</tr>
<tr>
<td>return addr</td>
<td></td>
</tr>
<tr>
<td>x: 4</td>
<td>ret: 24</td>
</tr>
<tr>
<td>return addr</td>
<td></td>
</tr>
<tr>
<td>x: 3</td>
<td>ret: 6</td>
</tr>
<tr>
<td>return addr</td>
<td></td>
</tr>
<tr>
<td>x: 2</td>
<td>ret: 2</td>
</tr>
<tr>
<td>return addr</td>
<td></td>
</tr>
<tr>
<td>x: 1</td>
<td>ret: 1</td>
</tr>
<tr>
<td>return addr</td>
<td></td>
</tr>
</tbody>
</table>
Tail Recursion

a1: 6, a2: 1
ret: 720

a1: 5, a2: 6
return addr

a1: 4, a2: 30
return addr

a1: 3, a2: 120
return addr

a1: 2, a2: 360
return addr

a1: 1, a2: 720
return addr
Code: gcc –O1

f2:

    movl   %esi, %eax
    cmpl   $1, %edi
    je     .L5
    subq   $8, %rsp
    movl   %edi, %esi
    imull  %eax, %esi
    subl   $1, %edi
    call   f2    # recursive call!
    addq   $8, %rsp

.L5:

    rep
    ret
Code: gcc –O2

f2:
  cmpl $1, %edi
  movl %esi, %eax
  je .L8

.L12:
  imull %edi, %eax
  subl $1, %edi
  cmpl $1, %edi
  jne .L12

.L8:
  rep
  ret
Quiz 1

What does this program do?

- a) repeatedly: reads a char, then writes it
- b) reads in all its input, then writes it out in the order it was read in
- c) reads in all its input, then writes it all out in reverse order
- d) reads in all of its input backwards, then writes it all out

```c
int main() {
    recur();
    return 0;
}

void recur() {
    char c = getchar();
    if (c != EOF) {
        recur();
        putchar(c);
    }
}
```
Reversing the Input

%rsp → abcdefghijklmnopqrstuvwxyz
Reversing the Input

%rsp → a

bcdefghijklmnopqrstuvwxyz
Reversing the Input

%rsp →

a
b
cdefghijklmnopqrstuvwxyz
Reversing the Input

\%rsp →

\[
\begin{array}{c}
a \\
b \\
c \\
\vdots \\
\end{array}
\]

\text{defghijklmnopqrstuvwxyz}
Reversing the Input

<table>
<thead>
<tr>
<th>a</th>
<th></th>
<th>qrstuvwxyz</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td></td>
<td></td>
</tr>
<tr>
<td>e</td>
<td></td>
<td></td>
</tr>
<tr>
<td>f</td>
<td></td>
<td></td>
</tr>
<tr>
<td>g</td>
<td></td>
<td></td>
</tr>
<tr>
<td>h</td>
<td></td>
<td></td>
</tr>
<tr>
<td>i</td>
<td></td>
<td></td>
</tr>
<tr>
<td>j</td>
<td></td>
<td></td>
</tr>
<tr>
<td>k</td>
<td></td>
<td></td>
</tr>
<tr>
<td>l</td>
<td></td>
<td></td>
</tr>
<tr>
<td>m</td>
<td></td>
<td></td>
</tr>
<tr>
<td>n</td>
<td></td>
<td></td>
</tr>
<tr>
<td>o</td>
<td></td>
<td></td>
</tr>
<tr>
<td>p</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Reversing the Input

print buf:

ponmlkjihgfedcba
int main() {
    char *buf;
    unsigned long cnt=0;
    long i;
    unsigned long ssize;

    for (ssize=16; ; ssize += 16) {
        Alloc16BytesOnStack(buf);
        // macro that modifies buf
        // to point to next 16 bytes
        for (i=15; i>=0; i--, cnt++) {
            if ((buf[i] =
                 getchar()) == EOF)
                goto done;
        }
    }
    done:
    write(1, &buf[i+1], cnt);
    write(1, "\n", 1);
    PopBytesOffStack(ssize);
    return 0;
}
Computer Architecture and Optimization (1)

What You Need to Know to Write Better Code
while (true) {
    instruction = mem[rip];
    execute(instruction);
}
void execute(instruction_t instruction) {
    decode(instruction, &opcode, &operands);
    fetch(operands, &in_operands);
    perform(opcode, in_operands, &out_operands);
    store(out_operands);
}
Analysis

• Not pipelined
  – each instruction takes, say, 3.2 nanoseconds
    » 3.2 ns latency
  – 312.5 million instructions/second (MIPS)

• Pipelined
  – each instruction still takes 3.2 ns
    » latency still 3.2 ns
  – an instruction completes every .8 ns
    » 1.25 billion instructions/second (GIPS) throughput
Hazards ...
Data Hazards

\[
\begin{align*}
\text{addq} & \ 12(\%rbx), \ %rax \\
\text{addq} & \ 20, \ %rax \\
\text{movq} & \ 40(\%rax), \ %rsp
\end{align*}
\]
Coping

Decode

12(%rbx),
%rax

addq

%rax

Decode

Decode

$20,
%rax

addq

%rax

40(%rax)

movq
Control Hazards

```assembly
movl $0, %ecx

.L2:
    movl %edx, %eax
    andl $1, %eax
    addl %eax, %ecx
    shrl $1, %edx
    jne .L2 # what goes in the pipeline?
    movl %ecx, %eax
    ...
```
**Coping: Guess ...**

- **Branch prediction**
  - assume, for example, that conditional branches are always taken
  - but don’t do anything to registers or memory until you know for sure
Modern CPU Design

**Instruction Control**
- Fetch Control
- Instruction Decode
- Instruction Cache
- Register File
- Retirement Unit
- Prediction OK?

**Execution**
- Integer/Branch
- General Integer
- FP Add
- FP Mult/Div
- Load
- Store
- Functional Units
- Operation Results
- Data Cache
- Addr.
- Data

M e m o r y
Performance Realities

There’s more to performance than asymptotic complexity

• Constant factors matter too!
  – easily see 10:1 performance range depending on how code is written
  – must optimize at multiple levels:
    » algorithm, data representations, functions, and loops

• Must understand system to optimize performance
  – how programs are compiled and executed
  – how to measure program performance and identify bottlenecks
  – how to improve performance without destroying code modularity and generality
Optimizing Compilers

• Provide efficient mapping of program to machine
  – register allocation
  – code selection and ordering (scheduling)
  – dead code elimination
  – eliminating minor inefficiencies
• Don’t (usually) improve asymptotic efficiency
  – up to programmer to select best overall algorithm
  – big-O savings are (often) more important than constant factors
    » but constant factors also matter
• Have difficulty overcoming “optimization blockers”
  – potential memory aliasing
  – potential function side-effects
Limitations of Optimizing Compilers

• Operate under fundamental constraint
  – must not cause any change in program behavior
  – often prevents it from making optimizations that would only affect behavior under pathological conditions
• Behavior that may be obvious to the programmer can be obfuscated by languages and coding styles
  – e.g., data ranges may be more limited than variable types suggest
• Most analysis is performed only within functions
  – whole-program analysis is too expensive in most cases
• Most analysis is based only on static information
  – compiler has difficulty anticipating run-time inputs

• When in doubt, the compiler must be conservative
Generally Useful Optimizations

- Optimizations that you or the compiler should do regardless of processor / compiler

- Code Motion
  - reduce frequency with which computation performed
    » if it will always produce same result
    » especially moving code out of loop

```c
void set_row(long *a, long *b, long i, long n){
    long j;
    for (j = 0; j < n; j++)
        a[n*i+j] = b[j];
}
```

```c
long j;
long ni = n*i;
for (j = 0; j < n; j++)
    a[ni+j] = b[j];
```
void set_row(long *a, long *b, 
long i, long n){ 
    long j; 
    for (j = 0; j < n; j++) 
        a[n*i+j] = b[j]; 
}

long j; 
long ni = n*i; 
long *rowp = a+ni; 
for (j = 0; j < n; j++) 
    rowp[j] = b[j];
Reduction in Strength

- Replace costly operation with simpler one
- Shift, add instead of multiply or divide
  \[ 16 \times x \quad \rightarrow \quad x \ll 4 \]
  - utility is machine-dependent
  - depends on cost of multiply or divide instruction
    » on Intel Nehalem, integer multiply requires 3 CPU cycles
- Recognize sequence of products

```c
for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
        a[n*i + j] = b[j];
```

```c
int ni = 0;
for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++)
        a[ni + j] = b[j];
    ni += n;
}
```
Share Common Subexpressions

- Reuse portions of expressions
- Compilers often not very sophisticated in exploiting arithmetic properties

```c
/* Sum neighbors of i,j */
up = val[(i-1)*n + j ];
down = val[(i+1)*n + j ];
left = val[i*n + j-1];
right = val[i*n + j+1];
sum = up + down + left + right;
```

3 multiplications: i*n, (i–1)*n, (i+1)*n

```c
long inj = i*n + j;
up = val[inj - n];
down = val[inj + n];
left = val[inj - 1];
right = val[inj + 1];
sum = up + down + left + right;
```

1 multiplication: i*n

```assembly
leaq   1(%rsi), %rax  # i+1
leaq   -1(%rsi), %r8  # i-1
imulq  %rcx, %rsi     # i*n
imulq  %rcx, %rax     # (i+1)*n
imulq  %rcx, %r8      # (i-1)*n
addq   %rdx, %rsi     # i*n+j
addq   %rdx, %rax     # (i+1)*n+j
addq   %rdx, %r8      # (i-1)*n+j
imulq  %rcx, %rsi     # i*n
addq   %rdx, %rsi     # i*n+j
movq    %rsi, %rax   # i*n+j+n
```

CS33 Intro to Computer Systems
Quiz 2

The fastest means for evaluating
\[ n^2 + 2n + 1 \]
requires exactly:

a) 2 multiplies and 2 additions
b) one multiply and two additions
c) one multiply and one addition
d) three additions
Optimization Blocker #1: Function Calls

- Function to convert string to lower case

```c
void lower(char *s){
    int i;
    for (i = 0; i < strlen(s); i++)
        if (s[i] >= 'A' && s[i] <= 'Z')
            s[i] -= ('A' - 'a');
}
```
Lower Case Conversion Performance

- Time quadruples when string length doubles
- Quadratic performance
Convert Loop To Goto Form

```c
void lower(char *s){
    int i = 0;
    if (i >= strlen(s))
        goto done;
    loop:
    if (s[i] >= 'A' && s[i] <= 'Z')
        s[i] -= ('A' - 'a');
    i++;
    if (i < strlen(s))
        goto loop;
    done:
}
```

- `strlen` executed every iteration
Calling strlen

```c
size_t strlen(const char *s) {
    size_t length = 0;
    while (*s != '\0') {
        s++;
        length++;
    }
    return length;
}
```

- `strlen` performance
  - only way to determine length of string is to scan its entire length, looking for null character
- Overall performance, string of length N
  - N calls to `strlen`
  - overall O(N^2) performance
Improving Performance

void lower2(char *s){
    int i;
    int len = strlen(s);
    for (i = 0; i < len; i++)
        if (s[i] >= 'A' && s[i] <= 'Z')
            s[i] -= ('A' - 'a');
}

• Move call to `strlen` outside of loop
  – since result does not change from one iteration to another
  – form of code motion
Lower-Case Conversion Performance

- Time doubles when string-length doubles
  - linear performance of lower2

![Graph showing the relationship between string length and CPU seconds for lower and lower2](graph.png)
Optimization Blocker: Function Calls

- Why couldn’t compiler move `strlen` out of inner loop?
  - function may have side effects
    » alters global state each time called
  - function may not return same value for given arguments
    » depends on other parts of global state
    » function lower could interact with `strlen`

- Warning:
  - compiler treats procedure call as a black box
  - weak optimizations near them

- Remedies:
  - use of inline functions
    » gcc does this with `-O2`
  - do your own code motion

```c
int lencnt = 0;
size_t strlen(const char *s){
    size_t length = 0;
    while (*s != '\0') {
        s++; length++;
    }
    lencnt += length;
    return length;
}
```
Memory Matters

/* Sum rows of n X n matrix a
   and store result in vector b */
void sum_rows1(long n, long a[][n], long *b) {
    long i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += a[i][j];
    }
}

# sum_rows1 inner loop
.L3:
    movq (%rdx,%rax,8), %rcx  # rcx = a[i][j]
    addq %rcx, (%rdx)         # b[i] += rcx
    addq $1, %rax             # j++
    cmpq %rax, %rdi           # if i<n
    jne .L3                   # goto .L3

• Code updates b[i] on every iteration
• Why couldn’t compiler optimize this away?
Memory Aliasing

/* Sum rows of n X n matrix a
   and store result in vector b */
void sum_rows1(long n, long a[][n], long *b) {
    long i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += a[i][j];
    }
}

int A[3][3] =
    {{ 0, 1,  2},
     { 4, 8, 16},
     {32, 64, 128}};
int *B = &A[1][0];
sum_rows1(3, A, B;

• Code updates b[i] on every iteration
• Must consider possibility that these updates will affect program behavior

Value of B:
init: [4, 8, 16]
i = 0: [3, 8, 16]
i = 1: [3, 22, 16]
i = 2: [3, 22, 224]
Removing Aliasing

/* Sum rows of n X n matrix a and store result in vector b */
void sum_rows1(long n, long a[][n], long *b) {
    long i, j;
    for (i = 0; i < n; i++) {
        long val = 0;
        for (j = 0; j < n; j++)
            val += a[i][j];
        b[i] = val;
    }
}

# sum_rows2 inner loop
.L4:
    addq (%r8, %rax, 8)), %rcx
    addq $1, %rdi
    cmpq %rcx, %rdi
    jne .L4

• No need to store intermediate results
Optimization Blocker: Memory Aliasing

• Aliasing
  – two different memory references specify single location
  – easy to have happen in C
    » since allowed to do address arithmetic
    » direct access to storage structures
  – get in habit of introducing local variables
    » accumulating within loops
    » your way of telling compiler not to check for aliasing
C99 to the Rescue

• New attribute
  – restrict
    » applied to a pointer, tells the compiler that the object pointed to will be accessed only via this pointer
    » compiler thus doesn’t have to worry about aliasing
    » but the programmer does ...
    » syntax
      int *restrict pointer;
Pointers and Arrays

- `long a[][n]`
  - `a` is a 2-D array of longs, the size of each row is `n`
- `long (*b)[n]`
  - `b` is a pointer to a 1-D array of size `n`

- `a` and `b` are of the same type
Memory Matters, Fixed

/* Sum rows of n X n matrix a
   and store result in vector b */
void sum_rows1(long n, long (*restrict a)[n], long *restrict b) {
    long i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += a[i][j];
    }
}

# sum_rows1 inner loop
.L3:
    addq (%rdi), %rax
    addq $8, %rdi
    cmpq %rcx, %rdi
    jne .L3

• Code doesn’t update b[i] on every iteration
Exploiting Instruction-Level Parallelism

• Need general understanding of modern processor design
  – hardware can execute multiple instructions in parallel

• Performance limited by data dependencies

• Simple transformations can have dramatic performance improvement
  – compilers often cannot make these transformations
  – lack of associativity and distributivity in floating-point arithmetic
Benchmark Example: Datatype for Vectors

/* data structure for vectors */
typedef struct{
    int len;
    data_t *data;
} vec_t, *vec_ptr_t;

/* retrieve vector element and store at val */
int get_vec_element(vec_ptr_t v, int idx, data_t *val){
    if (idx < 0 || idx >= v->len)
        return 0;
    *val = v->data[idx];
    return 1;
}

/* return length of vector */
int vec_length(vec_ptr_t v) {
    return v->len;
}
Benchmark Computation

```c
void combine1(vec_ptr_t v, data_t *dest)
{
    long int i;
    *dest = IDENT;
    for (i = 0; i < vec_length(v); i++) {
        data_t val;
        get_vec_element(v, i, &val);
        *dest = *dest OP val;
    }
}
```

Compute sum or product of vector elements

- **Data Types**
  - use different declarations for `data_t`
    - `int`
    - `float`
    - `double`

- **Operations**
  - use different definitions of `OP` and `IDENT`
    - `+`, `0`
    - `*`, `1`
Cycles Per Element (CPE)

- Convenient way to express performance of program that operates on vectors or lists
- Length = n
- \( T = \text{CPE} \times n + \text{Overhead} \)
  - CPE is slope of line

\[ \begin{align*}
\text{vsum1:} & \quad \text{Slope} = 4.0 \\
\text{vsum2:} & \quad \text{Slope} = 3.5
\end{align*} \]
Benchmark Performance

```c
void combine1(vec_ptr_t v, data_t *dest){
    long int i;
    *dest = IDENT;
    for (i = 0; i < vec_length(v); i++) {
        data_t val;
        get_vec_element(v, i, &val);
        *dest = *dest OP val;
    }
}
```

Compute sum or product of vector elements

<table>
<thead>
<tr>
<th>Method</th>
<th>Integer</th>
<th>Double FP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation</td>
<td>Add</td>
<td>Mult</td>
</tr>
<tr>
<td>Combine1</td>
<td>29.0</td>
<td>29.2</td>
</tr>
<tr>
<td>unoptimized</td>
<td>27.4</td>
<td>27.9</td>
</tr>
<tr>
<td>Combine1 –O1</td>
<td>12.0</td>
<td>12.0</td>
</tr>
</tbody>
</table>
Move \textit{vec\_length}

\begin{verbatim}
void combine2(vec_ptr_t v, data_t *dest) {
    long int i;
    long int length = vec_length(v);
    *dest = IDENT;
    for (i = 0; i < length; i++) {
        data_t val;
        get_vec_element(v, i, &val);
        *dest = *dest OP val;
    }
}
\end{verbatim}

<table>
<thead>
<tr>
<th>Method</th>
<th>Integer</th>
<th>Double FP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Add</td>
<td>Mult</td>
</tr>
<tr>
<td>Operation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Combine1 unoptimized</td>
<td>29.0</td>
<td>29.2</td>
</tr>
<tr>
<td>Combine1 –O1</td>
<td>12.0</td>
<td>12.0</td>
</tr>
<tr>
<td>Combine2</td>
<td>8.03</td>
<td>8.09</td>
</tr>
</tbody>
</table>
Eliminate Function Calls

```c
void combine3(vec_ptr_t v, data_t *dest) {
    long int i;
    long int length = vec_length(v);
    data_t *data = get_vec_start(v);
    *dest = IDENT;
    for (i = 0; i < length; i++) {
        *dest = *dest OP data[i];
    }
}
```

data_t *get_vec_start(vec_ptr v) {
    return v->data;
}

<table>
<thead>
<tr>
<th>Method</th>
<th>Integer</th>
<th>Double FP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation</td>
<td>Add</td>
<td>Mult</td>
</tr>
<tr>
<td>Combine2</td>
<td>8.03</td>
<td>8.09</td>
</tr>
<tr>
<td>Combine3</td>
<td>6.01</td>
<td>8.01</td>
</tr>
</tbody>
</table>
Eliminate Unneeded Memory

References

```c
void combine4(vec_ptr_t v, data_t *dest) {
    int i;
    int length = vec_length(v);
    data_t *d = get_vec_start(v);
    data_t t = IDENT;
    for (i = 0; i < length; i++)
        t = t OP d[i];
    *dest = t;
}
```

<table>
<thead>
<tr>
<th>Method</th>
<th>Integer</th>
<th></th>
<th>Double FP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Add</td>
<td>Mult</td>
<td>Add</td>
</tr>
<tr>
<td>Operation</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Combine1</td>
<td>12.0</td>
<td>12.0</td>
<td>12.0</td>
</tr>
<tr>
<td>–O1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Combine4</td>
<td>2.0</td>
<td>3.0</td>
<td>3.0</td>
</tr>
</tbody>
</table>
Quiz 3

Combine4 is pretty fast; we’ve done all the “obvious” optimizations. How much faster will we be able to make it? (Hint: it involves taking advantage of pipelining and multiple functional units on the chip.)

a) 1× (it’s already as fast as possible)
b) 2× – 4×
c) 16× – 64×