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()`

```c
/* 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.
Supplied by CMU, but adapted for x86-64.
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.
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).
Buffer Overflow Example #2

Before call to gets

Input 123456789ABCDEF01234567

Stack frame for main

00 00 00 00 40 05 73
00 37 36 35 34 33 32 31
30 46 45 44 43 42 41 39
38 37 36 35 34 33 32 31

Stack frame for main

[Image of stack frames]

Still no problem

40056e: e8 d9 ff ff ff callq 40054c <echo>
400573: b8 00 00 00 00 mov $0x0,%eax

Supplied by CMU, but adapted for x86-64.
Buffer Overflow Example #3

Before call to gets

Stack frame for main

Return Address

Input 123456789ABCDEF012345678

Stack frame for main

00 00 00 00 40 05 00
38 37 36 35 34 33 32 31
30 46 45 44 43 42 41 39
38 37 36 35 34 33 32 31

Return address corrupted

40056e: e8 d9 ff ff ff callq 40054c <echo>
400573: b8 00 00 00 00 mov $0x0,%eax

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

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

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.
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 0x7fffffff948. What value should be put into the return-address portion of the stack frame?

a) 0
b) 0x7fffffff948
c) 0x7fffffff9a0
d) it doesn’t matter what value goes there
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.
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 (0xfffffffffe948) 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.
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.
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.
Again we have the output from “objdump –d”.

```assembly
Again we have the output from "objdump –d".
```
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):

```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`
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.
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.

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

Before call to gets

Stack frame for main

Return address

Canary

buf:  [3] [2] [1] [0]

/* 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.
Checking Canary

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

buf
[3][2][1][0]

$rsp$

CS33 Intro to Computer Systems

Supplied by CMU.