Introduction (0:00-2:30)

- Today we welcome special guest Dean of the Faculty of Arts and Sciences Mike Smith
- Before becoming Dean of FAS, Professor Smith was the instructor of cs50!
- He will talk to us today about how people can exploit weaknesses in our code and how we try to defend against them

Dangerous Functions (2:30-7:30)

- Many of the functions we use regularly can produce weaknesses in our code if not used correctly
- In this class, we are accustomed to using the GetString and CopyString functions from the cs50 library to get text input from the user
- But these functions, as you have learned, are dangerous because they have memory leaks.
- The ANSI library (string.h) also contains functions for handling strings, such as strcat, strcpy, strcmp, and strstr
- But these functions are also not ideal. We use them because they're efficient and easy-to-use, but they don't do any checking on buffer size and therefore can be dangerous.
- Some other dangerous functions from file i/o: getc, scanf, printf, fprintf

What Is Secure Code? (7:30-11:30)

- Code without comments? No. This obfuscates code, but the adversary can still figure out what it does and attack it.
- Code that uses cryptography? No. Your code may implement security features, but not within a well-formed, secure framework. For instance, a cryptographic scheme is difficult to break without knowledge of the secret key, but carelessness with memory could make it easy to locate the key.
- Code that is hard to use? No. It might take the adversary a bit more time, but a determined adversary will not be deterred by that.
- Code written in Java or another newer language? No. These new languages help to avert some of the problems we'll see, but are not sufficient.
- What we really mean is code designed to be robust and withstand attacks by malicious users.
- In order to defend against the adversary, we need to think like the adversary

Realities (11:30-13:30)

- If you are writing code that will be run on a network, you should assume that there are adversaries with extensive resources attempting to attack it
- Security should influence your design from the start, not be something you attempt to tack on at the end

- You should also assume that your code will always have bugs. Bugs are what allow adversaries to take advantage of our code. We should anticipate these security failures from the start

Implementing a Buffer Overrun Attack (13:30-23:00)

- So how do people take advantage of our code?
- In a buffer overrun attack, the adversary tries to put program control in a place where it shouldn't be. Then they can take control of the program in that new location and make it do what they want.
- They will achieve this goal by writing to memory beyond the end of a buffer.
- C is inherently weak because it does not have a native string type and a corresponding set of safe, easy-to-use functions.
- As a result, it is the programmer's responsibility to do all the checking when using strings.
- Careless coding that does not include these checks is vulnerable to attack
- "Buffer overflows accounted for more than 50% of all major security bugs resulting in CERT/CC advisories in 1999, and the data show that the problem is growing instead of shrinking." Viega and McGraw, p. 135.
- Even now, a huge number of the attacks that occur are a result of buffer overflow
- How does a buffer overrun work?
- Consider the following code snippet:

```
int
main(int argc, char * argv[])
{
    int i;
    for (i = 1; i < argc; i++)
         printf("%s ", argv[i]);
    printf("\n");

    return 0;
}
```

- This prints out each of the command line arguments
- We can do the same thing by using a function called echo_arg:

```
int
main(int argc, char * argv[])
{
    int i;
    for (i = 1; i < argc; i++)
         echo_arg(argv[i]);
    printf("\n");

    return 0;
}
```

- echo_arg is then implemented as follows:

```
void
echo_arg(const char s[])
{
char buf[MAX_BUF_SIZE];
strcpy(buf, s);
printf("%s ", buf);
}
```

- It takes a string, copies it into a buffer, and prints out the contents of the buffer.
- Will this code work? Yes.
- What's the problem?
- When we declare the local variable buf, it gets put on the stack. Suppose MAX_BUF_SIZE is #define-ed to be 4. Then buf is a char array of length 4 on the stack.
- What if we enter a string longer than 4 characters? strcpy keeps on copying until it encounters a null character in s.
- Now, we compile eco.c into the executable eco and try to run it
- Suppose we run `./eco ls -l foo`. Then it prints out ls -l foo. There is no problem because every argument is three or fewer characters.
- But when we run `./eco make foo`, it prints out make and then seg faults because make is a 5-character array.
- So we've broken the program by feeding it particular inputs.
- This is one way to attack software called denial of service. We repeatedly crash the program, making other users get annoyed and go elsewhere.
- But let's say we want to attack in such a way that we can also take control of the program.

Redirecting Program Control (23:00-33:00)

- Consider the following version of eco.c:

```
void
gotcha()
{
    printf("\nGotcha!\n");
    exit(1);/* required because we destroy the caller's base pointer */
}


void
echo_arg(const char s[])
{
    char buf[MAX_BUF_SIZE];
    strcpy(buf, s);
    printf("%s ", buf);
}


int
main(int argc, char * argv[])
```

```
{
    int i;
    for (i = 1; i < argc; i++)
         echo_arg(argv[i]);
    printf("\n");

    return 0;
}
```

- Will it ever run gotcha()? Seemingly not, because there's no call to it anywhere in the code.
- But when we run `./hack.pl ./eco`, we get this output: 1234^X\277\374\203 Gotcha!
- So what did hack.pl do? hack.pl is a Perl program that creates a variable called arg containing 1234 concatenated with a bunch of hex characters. Then we execute ./eco with this contents of this variable arg as our argument
- We do this in order to get some unprintable characters into argument
- How does this work? Here's what we need:
    - Fixed size buffer declared on the stack
    - Unsafe function to write the contents of the buffer – that is, a function that will allow us to write beyond the end of the buffer
    - Make sure nobody checks the content of what we send to this buffer – that's what allows us to send in something that doesn't look like a character
    - Adversary's input changes return address of a function on the stack – that's what allows us to get into a function that wasn't other wise called
- When we start running a program, a stack frame gets created for main at the bottom of the stack. It contains all of main's local variables.
- When we call echo_arg, a stack frame gets created for echo_arg above main on the stack. It contains buf.
- When echo_arg gets closed, that stack frame will get erased and we'll return to main. How do we know where in main go back to? When echo_arg is called, main stores a return address that tells it where to come back to after echo_arg gets popped off.
- Now, think back to buf. If we write past the bounds of buf, we move "down" the stack. (We are actually moving toward high memory, but the way we draw our picture, we move down the stack toward where main is.)
- This means that if we write off the end of buf far enough, we can actually overwrite the return address!

Studying the Stack (33:00-48:00)

- We can study the stack with the following print statement, contained in smash.c:

```
printf("\nContents of echo_arg's stack BEFORE strcpy:\n"
     "0x%08x -- garbage\n"
     "0x%08x -- garbage\n"
     "0x%08x -- garbage\n"
```

```
"0x%08x -- garbage\n"
"0x%08x -- initial contents of buf\n"
"0x%08x -- contents of EBP in main\n"
"0x%08x -- return address for call to echo_arg\n"
"0x%08x -- address of string passed to echo_arg\n");
```

- %08x is a format string (just like %s or %d) means we are printing something in hex in eight zero-filled digits. We precede the output with 0x to indicate that what follows is a hex number.
- Normally when we use a format sting, we give printf variables. These variables get put on the stack and then taken off by printf as it finds the need for variables in the string.
- Since we don't give it variables, it just goes and takes whatever's on the stack. This is how we can print the contents of the stack.
- We use this print statement to see what's on the stack before and after strcpy when we run `./smash ls`
- What we find is that the contents of buf contains garbage before strcpy and ls after strcpy. This is good.
- We also see that the return address for the call to echo_arg does not change. It's some place in main.
- Now we run `./hack.pl smash`
- This time, we get a Gotcha! with the junk as before
- We also see that the final contents of buf are 1234 followed by some other characters, and that the return address for call to echo_arg has changed. In fact, it is now the base address of gotcha!
- So when echo_arg closes, we don't go back to main. We go to gotcha.
- At the end of gotcha, we have an exit, so the program stops after the print statement.
- Question: how do we know the address of gotcha and how do we write just enough characters to buf to get that address?
- In this case, Prof. Smith did it using the debugger.
- If this was not an option, you could just keep sending random addresses (using some automated process) until something works
- How can you prevent buffer overruns?
    - Always validate your inputs
    - Never use strcpy (use strncpy)
    - Fail gracefully
- Another similar exploit is the heap overrun, in which we do this on the heap instead of the stack
- That's all we have time for here, but check out Prof. Smith's slides to see how strcpy, gets, printf, and sprintf can all be unsafe functions. Also see some suggested reading on his last slide.