CSE 409, Fall 2011, Rob Johnson, http://www.cs.stonybrook.edu/~rob/teaching/cse409-fa11/
Alin Tomescu, September 28th, 2011

Examples of vulnerable code

Stack overflow prevention mechanisms

There are a couple of mechanisms Linux and other operating systems use to protect against stack overflow attacks.

Address space layout randomization

The kernel **randomizes the location of the stack** when your program runs, rather than having the stack always begin at a fixed address.

There is a paper (this one?) that shows that in about two hours you might be able to overflow the stack if you keep trying.

Non-executable stack

The kernel prevents the program from executing code from the stack by **marking the stack pages as non-executable**. This can be turned off by the user, which would be a very bad idea.

Canary stack protector

Remember: When your program makes a function call it pushes the function's arguments and the return address (in this order) on the stack. Then the called function pushes its local variables below the return address.

0xFFFF	**Stack, grows downward**
	Argument 1
	Argument 2
	Return address
	Local variable 1
	Local variable 2
0x1000	etc.

With stack protection, a **random value** known as a **canary** is introduced between the return address and the function's local variables. This way, an attacker cannot overwrite the return address without overwriting the canary. The program can tell if the canary was modified and a buffer overflow occurred since it has a copy of the canary on the heap or in the global section somewhere.

Note:
- This is all done by the program so it's a **feature of the compiler** and not the kernel.
- With a **format string attack**, we can skip over the canary and write the return address.
- This will add overhead to your program, since 4 extra bytes are pushed onto the stack for every function call

The stack protector is compiled into the program by default unless the user specifies the `–fno-stack-protector` flag.

Shell code injection attack

The stack described in class is a little simpler than in reality:

```
struct stack {
    char buf[512];
    char pad[8];
    void * saved_ebp;    //  Old stack pointer
    void * retaddr;
} __attribute__((packed));

//  Fill out top half of the buffer with argument to system() also w/ landing strip
(spaces)
//  Fill out bottom half of the buffer w/ shell code and landing strip (no ops)

//  ebp replaced with some value
//  retaddr made to point somewhere in the middle of the bottom half of the buffer
```

Note: The `system` libc function actually looks on the stack at the return address to determine if it was called by code executing on the stack. If it was, then `system` refuses to execute. In our shellcode, we trick `system` by setting its return address to some random address in the .text segment.

Return-to-libc attack

The code:

```
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <sys/mman.h>
#include <assert.h>
#include <stdlib.h>

int parse_file(char * arg)
{
    char buf[512];

    strcpy(buf, arg);

    return 0;
}

int main(int argc, char ** argv)
{
    int fd;
    char * contents;
    struct stat sb;

    fd = open(argv[1], O_RDONLY);
    fstat(fd, &sb);

    contents = malloc(sb.st_size + 1);
    read(fd, contents, sb.st_size);
    contents[sb.st_size] = 0;

    parse_file(contents);

    return 0;
}
```

If we're going to do a return-to-libc attack on `system`, then we need to setup the stack for `system`.

CSE 409, Fall 2011, Rob Johnson, http://www.cs.stonybrook.edu/~rob/teaching/cse409-fa11/
Alin Tomescu, September 28th, 2011
The stack for system:

```
struct stack {
    char buf[512];
    char pad[8];
    void * saved_ebp;
    void * retaddr;
    void * systemretaddr;   //  system ret addr
    char * systemarg;       //  system argument
} __attribute__((packed));

//  Fill out top half of the buffer with argument to system() also w/ landing strip
(spaces)
//  Fill out bottom half of the buffer w/ shell code and landing strip (no ops)

//  ebp replaced with some value
//  retaddr made to point somewhere in the middle of the bottom half of the buffer
```

If a normal function would call `system`, then `system` would expect to find the return address and its command argument on the stack. Also, another concern is that after we jump from `parse_file` to `system`, then `system`'s local variables will overwrite our `parse_file` buffer which was on the stack. We therefore can't store the argument for `system` in that buffer.

Since the initial program `malloc`'d the file it read, we can point to `system`'s argument inside the `malloc`'d buffer.

This time we don't execute code on the stack, so this attack will work an OS with non-executable stack protection. We just jump to system which gets its arguments from the stack we "fixed up" using the buffer overflow.

Format string attack

Problem: If you want to set the value of an address to something like 0xFFFF FFFF you would have to do a `snprintf("%4294967296d")` which would mean `snprintf` would have to generate gigabytes of data. This is highly impractical since it would take a lot of time and space.

Refresher on endianness

Little endian: The least significant byte is stored in the lower address. Intel CPU's are little endian.
Big endian: The most significant byte is stored in the lower address.

On a little endian machine, if I want to write ABCD (4 bytes) at location x, then I am going to get `printf`'s count to be equal to $0x000D$ (or any other value which has the last byte set to D) and write it to address x.

Then, I'll change the count to $be\ 0x000C$ and write it to the next address $x + 1$. I can keep this up until byte A and I'll end up filling the memory with the value I wanted at the address I wanted.

The downside is that 3 extra bytes are clobbered on the higher addresses, since the last write for $0x000A$ will zero out the 3 bytes after the A, so addresses $x + 4, x + 5$ and $x + 6$ will be zeroed.