6.033: Security - Low-level Exploits
Lecture 21
Katrina LaCurts, lacurts@mit.edu

0. Introduction
   - Today + tomorrow's recitation: low-level attacks
   - Motivation: how might an attacker without sysadmin privileges gain access to stored data (or do other things on a machine)?

Part 1: Buffer overflows ("Stack smashing")

1. Set up
   - What's stored on the program stack when we call a function
     - local variables from the calling function
     - the arguments to functions
     - saved base pointer, saved instruction pointer
     - local variables within the function
   - Important point: saved pointers let us return to the right place in the calling function
   - Goal of these attacks (in lecture is to exploit the gets() function, which requests user input and stores it in a pre-allocated buffer. We're going to write past the pre-allocated bytes.

2. Attack 1: overwrite a variable (see slides for code)
   - If we use a 64-byte string (or a smaller one), nothing gets overwritten
   - Adding a 65th byte overwrites the variable modified, because it's stored next on the stack
   - C does not protect against this!

3. Attack 2: overwrite a more interesting variable (a function pointer)
   - Same overwrite from before works, but sends us to a nonsensical place
   - We can set fp to point to win() as long as we know the address where it's stored. We can get that from gdb (the GNU DeBugger)
     - We just have to take care to give the address input in the right byte order

4. Attack 3: overwrite saved state on the stack
   - What if there are no useful variables to overwrite? Then we can overwrite the saved instruction pointer
   - Slightly more complicated: have to figure out where instruction pointer is saved in relationship to our buffer
     - There might be some other stuff in between them; it just depends on the specifics of how an architecture's stack works
   - This is (relatively) easily-doable with the help of gdb

5. Further attacks
   - The next step in this process would be to execute our *own* code instead of a function like win. Typically we want to execute code that opens a shell, because from there we can do all sorts of nefarious deeds

6. Counter-attacks / Solutions
   - Modern linux protects against all of this, but there are still counter-attacks
   - Solution: non-executable stack
   - Solution: Address space layout randomization (randomize where things are stored)
   - Counter: "arc injection" attacks exploit parts of memory that can't easily be randomized, and doesn't require an executable stack
     - These attacks effectively use libc to start a shell
   - Solution: save pointers to the heap too, and compare, to check for overwrites
   - Counter: heap-smashing. Works differently than stack-smashing, but has similar results
   - Counter: "pointer subterfuge" attacks don't require the saved pointers to be overwritten at all
   - Bounds-checking would be the best solution, but the performance cost is high.

Part 2: Compilers ("Trusting Trust")

7. Compilers
   - Input source code; output machine code
   - The C compiler can compile itself
     - Seems weird, but this is not that uncommon in the world of compilers

8. Inserting backdoors
   - If adversary adds a hack to the UNIX src, people will know
   - If adversary instead adds a hack to the C compiler, people will still know; they can read the compiler src
     - In the compiler, the "hack" is "if compiling the UNIX src, then insert backdoor"
   - What if the adversary lies about the src?
     - Can still detect the problem by recompiling the clean source
   - Change our hack: the hacked compiler now inserts a backdoor into UNIX *and* into the C compiler
     - In the compiler: "if compiling the UNIX src then insert backdoor; if compiling the C compiler then insert this backdoor-inserting code"
   - Now the lie-detection attempt -- where we recompile the compiler -- doesn't work

9. History

- This attack comes from Ken Thompson, in his Turing Award acceptance speech
- The speech advocates for policy changes, not technology-based solutions