Chapter 5: Buffer Overrun

→ Formerly public enemy number one (PHP exploits appear to have been number one for 2006)

1. Buffer overruns are a problem because of poor coding practices, specifically in C and C++
    a. Memory must be managed by programmer (NOTE: this can be overcome somewhat in C++ by utilizing the STL)
    b. They lack safe (and easy to use) string handling functions (once again, STL in C++ can mitigate this)
        i. Stay away from strcpy, strcat, sprintf, gets, (even strncpy is vulnerable)
        ii. Use fgets to read strings, find a library of safe string functions if possible (strsafe.h by M$), or write your own!
    c. No bounds checking on arrays (C++ STL helps here, too)
    d. IF you are given a choice, try to avoid C for complex/big software development
        i. Most OSs are C based
        ii. Many compilers and interpreters are written in C (and are thus themselves vulnerable to overruns)
2. Stack-based overruns: occurs when a buffer declared on the stack is overwritten by copying data larger than the buffer
    a. Usually occurs due to unchecked user input (see ch. 10: All Input is Evil)
    b. Result is execution of arbitrary/malicious code
    c. Often times, stack frame has return address overwritten (which leads to execution of arbitrary code) (can overwrite to return to the start of the buffer – which contains the attack payload)
    d. See: Smashing the Stack for Fun and Profit, by Aleph One: http://insecure.org/stf/smashstack.html
    e. "A buffer overrun is exploitable, unless it's not" – avoid them, watch out for them, and certainly fix them at any cost
    f. Compiler mitigation
        i. Canaries GS (Guard Stack)
        ii. Make stack contents static (umm, usually not practical)
        iii. Reverse order of where things are placed on the stack (make the return address lower address than the data section – then buffer underflow is required)
    g. Some OSs and chips can be configured to have a non-executable stack (no code on the stack can be executed directly)
3. Heap-based overruns: same idea as with a stack – arbitrary information is written to your program space on the heap (dynamically allocated memory)
    a. Can overwrite into another dynamically allocated variable (what if the next variable holds a filename?)
    b. Harder to exploit than stack, but still possible
4. Array Indexing Errors: not as common as buffer overruns

    a. Same concept as buffer overruns – you write past the bounds of an array and overwrite other program memory and get arbitrary code to execute
5. Truncation errors
    a. Large values can be truncated based on where they are stored (word size)
    b. Signed/unsigned issues can lead to truncation (and thus an undesired value that is used to access some memory location)
6. Format string bugs: a function that takes a variable number of arguments does not have a failsafe way to determine beyond a doubt how many arguments were actually passed in
    a. printf( ) family does this
    b. memory space can once again be overwritten by malicious code based on how data is read in and translated based on format specifiers of the function
    c. you should always specify a format string when using the printf( ) family (don't just say: printf(value);)
7. Watch for size mismatches, they can be exploited
    a. Unicode (2 bytes) vs. ASCII/ANSI (1 byte)
    b. Checking size based on one type then using that size on the other type can lead to arbitrary code execution
8. PREVENTION
    a. Always validate input – make no assumptions!
    b. Avoid dangerous built-in functions (strcpy( ), etc.)
    c. Use safe alternatives to the above (C++ STL, strsafe.h – included in code for book)
    d. Use compiler options if available
    e. Use a third party product if necessary (StackGuard)
    f. → Write good, clean, secure code