Buffer Overflows: Attack and Defense

- Buffer overflow vulnerabilities are the most common way to gain control of a remote host
- Most common security vulnerability
- Buffer overflow vulnerabilities are common and easy to exploit
- Attacker can insert and execute attack code
- Error is made at program creation, is invisible to user
Definition: Exploit

- An exploit is code that takes advantage of a bug in other code
- Exploits can cause:
  - Crash of machine and denial of service
  - Crash of just the program that is running
- Exploits have two halves
  - Injection vector – uses the bug itself to get in
  - Payload – not related to the bug at all, can do just about anything an attacker wants to do
Exploit Injection Vector

- Depends totally upon the operating system as well as the target hardware platform
- Content based injector inserts characters in the data that result in the program doing a bad thing but the process is still cognizant
- Buffer Overflow based injector causes process to lose cognizance by overflowing the stack, overflowing the heap, causes an instruction pointer to go to an attacker controlled value in order to point to an attack controlled buffer
Exploit Payload

- Independent of Exploit Injection Vector
- Dependent upon hardware platform, operating system
- Much like a virus
- Can cause remote shell, can cause rootkit to be installed, can cause worm or virus activity, can cause a denial of service
Buffer Overflow Principle

- Overwrite parts of memory that are not intended to be overwritten
- Make process execute this overwritten memory
Computer Memory Usage

- Code Segment – Assembly instructions processor executes
- Data segment – Variables and buffer
- Stack Segment - Store data to pass to functions and storage for function variables
Computer Memory Usage

Local Variables

- Attack Code
- Return Address
- Local Variables
- Buffer

String fills this way

Stack fills this way
Example Code

```c
void function(char *str) {
    char buffer[16];
    strcpy(buffer,str);
}

void main() {
    char large_string[256];
    int i;
    for( i = 0; i < 255; i++)
        large_string[i] = 'A';

    function(large_string);
}
```

(Source: “Smashing the Stack”)
Fundamental Concept

• By overwriting the return address with a value that points to attack code, can cause attack code to execute!
Attacker Goals

• To put or to use “opportunistic” code into program’s address space
• Cause the execution of the program to jump to the “opportunistic” code
Putting or Using “Opportunistic” code

- Insert the “opportunistic” code by string input to program which is written to buffer
- Use “opportunistic” code that already exists and executes functions like exec(argument) where one can pass the argument /bin/sh to cause exec(“/bin/sh”)
How To Get The Program to Jump

- Corrupt the return address stored on the stack
- Most common technique
  - Give a program a large string that both overflows the buffer to overwrite the return address and to put the attack code into program memory
Defense one

- Correct the source code in the programs
  - Check the source code for fgets, gets, getws, memcpy, memmove, scanf, sprintf, strcat, strncpy where the length of the arguments are not checked
  - Use fault injection tools and/or static analysis tools
Defense 2

• Make the data segment of the victim program address space non-executable
• Kernel patches exist to do this
• “almost” no programs have code in the stack segment
Defense 3

• Array bounds checking – all reads and writes need to be checked to make sure they are within range
  ▪ If arrays cannot be overwritten, then no buffer overflow exploits
  ▪ Hard to do this in compilers
Defense 4

- Code Pointer Integrity Checking – check to see if a return value has been corrupted before using it
  - Example is StackGuard which puts a “canary” word next to the return address in the stack

(Source: “Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade”
http://www.immunix.org/StackGuard/discex00.pdf)