HOW DO THEY DO IT?
Buffer Overflows

"An error the breadth of a single hair can lead one a thousand miles astray."

Chinese Proverb

Introduction

- The most insidious computer attacks that we know of are discussed in this chapter!
- They exploit vulnerabilities that you have little control of
- They are incredibly difficult to discover and fix
- They are vulnerabilities built into commercial software apps like, IIS, Oracle DB servers and Sun’s Java Web server
- You cannot find and fix yourself!

Simple Buffer Overflows

- Simply, a buffer overflow occurs when the amount of data being written to memory is larger than the amount of memory reserved for the operation
- When this happens, the data being written usually gets written beyond the reserved section
- The data has to go somewhere...

Simple Buffer Overflows

- Consider:

```c
void overflow ( void )
{
    char *name = “hackingexposedhackingexposed”;
    char buff[10];
    strcpy ( buff, name );
    return
}
```

- buff is allocated 10 bytes
- name is copied into buff
- name is larger than buff and the data overflows into memory

Simple Buffer Overflows

- But where did the extra data go?
- How can we use those rogue bytes to our advantage?
- To answer those Q’s, we need to discuss assembly language!
- The discussion will be general to all assembly languages but will use the terminology of x86

Assembly in a Nutshell

- Assembly Language (AL) is a low-level language written to a particular architecture and CPU
- Numerous variations: Intel x86, SPARC, RISC, etc.
- AL allows programmers “direct” access to certain pieces of hardware
  - Serial ports
  - Memory
  - Graphics cards
  - Etc.

Assembly in a Nutshell

- AL is the computer’s “native tongue”
- You are programming with the CPU’s actual instructions
Memory is a string of binary bits organized (for the user) into bytes, typically 8 bits
This allows us to represent non-binary numbers in memory
The CPU contains small, memory-like, storage units called registers

**Assembly in a Nutshell**
- Registers can come in a variety of sizes: 8, 16, 32 and 64 bits
- General purpose registers are generally 16 bits and named AX, BX, CX & DX
- The programmer can use the general purpose registers however they like
- Pre 80386 days, AX was made up of 2 8-bit halves: AL and AH
- Same for BX, CX & DX
- In the 32-bit world, EAX is 32-bit extended AX

**Assembly in a Nutshell**
- The general purpose registers are sometimes used for special purposes:
  - EAX = The accumulator. Primarily used for I/O, arithmetic and calling services
  - EBX = Base register. Used as a pointer to a base address
  - ECX = Count register. Used in looping
  - EDX = Data register

**Assembly in a Nutshell**
- Pointer Registers:
  - Usually used for string instructions
  - 3 pointer registers & 2 index registers
    - ESP = Stack pointer
    - EBP = Base pointer (points into the stack)
    - EIP = Instruction pointer, next instruction
    - ESI = Source index
    - EDI = Destination index

**Assembly in a Nutshell**
- The Stack
  - Special segment in memory used for:
    - Store the return addresses of functions
    - Store register values
    - Store variables
  - Like a pile of plates: LIFO data structure
  - Push data on, pop data off
  - When a function is called, the register values, function parameters and the return address are pushed onto the stack

**Tracking the Rogue Bytes**
- Okay, great, but what does this have to do with buffer overflows?
- Let's go back to our original example and see how the computer executes the program

```c
void overflow ( void )
{
    char *name = "hackingexposedhackingexposed";
    char *buff[10];
    strcpy ( buff, name );
    return
}
```

**Tracking the Rogue Bytes**
void overflow ( void )
```c

void overflow ( void )
{
    char *name = "hackingexposedhackingexposed";
    char buff[10];
    strcpy ( buff, name ); ←
    return
}

When overflow() is called, the stack looks like this at the point shown by the arrow:
    address of *name
    address of buff
    vars
    buff
    saved EBP
    return address

Note that buff is allocated on the stack

The extra bytes written to buff overflow & overwrite the saved EBP and the return address!

When overflow returns, what is popped off the stack? A: what should be the EBP and what should be the return address

The instruction at 0x70786567 referenced memory at 0x70786567. The memory could not be read

Let’s look at the error a little closer...
• 0x70786567? What is this?
• It’s, "pxeg" as in “hackingexposed”
• Thus, we can overwrite the EIP with the strcpy() of buff
• If we can alter the EIP, we can seriously alter the course of the program by pointing the EIP code to code that we overwrite in memory by overflow!
• What if we overfill the buffer with machine code?
```
Let’s look at the error a little closer…
- What if we overfill the buffer with machine code?
- We have the single greatest fear of any security team: execute any command on the target system without ever guessing a password!
- The world is our oyster!

**Buffer Overflow: An Example**
- First of all, we have 3 options in our quest for buffer overflow vulnerabilities:
  1. Source code review
  2. Disassembly
  3. Blind stress testing
- With Windows exploits, we are limited to #2 & #3
- Let’s take a look at each...

**Buffer Overflow: An Example**
- Disassembly: the art of taking a binary executable program and turning it into assembly language or instructions for the CPU to carry out
- There are a number of disassemblers available
- This ability is valuable in the closed-source world of Windows
- To locate vulnerabilities via disassembly requires a knowledge of how functions are translated into asm.

**Buffer Overflow: An Example**
- Consider:
  ```c
  int vuln ( char *user ) {
      char buffer[500];
      sprintf ( buffer,
          "%s is an invalid username",
          user );
      return 1;
  }
  ```
- An argument of user-defined length user is being copied into the 500 byte length buffer

**Buffer Overflow: An Example**
- But, we control the length of user so we can overflow it and execute arbitrary code
- Disassembled code:
  ```assembly
  mov eax, [ebp+8]
  push eax
  push offset aSIsAnInvalidUs
  lea ecx, [ebp-1F4h]
  push ecx
  call _sprintf
  ```

**Buffer Overflow: An Example**
- Some essentials:
  - Parameters are pushed on the stack in reverse order
  - (Almost) every instruction that references memory above EBP (e.g., [ebp+8]) is referencing a procedure parameter
  - Local variables are referenced as negative offsets from EBP (e.g., [ebp-1F4h])

**Buffer Overflow: An Example**
- So, what does our disassembled code do?
  - Our user parameter is moved to EAX and pushed on the stack
• The sprintf arguments are pushed on the stack
• buffer is pushed on the stack (0x1f4 is 500 decimal)

We need to keep in mind the essentials from the last slide... as we’ll see!

25 Buffer Overflow: An Example

The next part is somewhat involved so I’ll try to summarize...
• Let’s say this code is running on the server:
  ```c
  void crash ( void ) {
    char buff[1400];
    strcpy ( buff, rbuff );
    return;
  }
  ```
• The data in rbuff is received via a socket so by pushing a bunch of bytes into rbuff will overflow buff

26 Buffer Overflow: An Example

Summery (cont)
• We set a breakpoint in the Windows exception handler (via SoftIce, a kernel mode debugger for Windows)
  ○ This mean that when Windows throws an exception, SoftICE will break
  ○ This signals our “arbitrary code”
• Next, we fling 1400 bytes of x’s at the open port plus “abcd”. We could use NetCat to do this.
• SoftIce shows that the buffer overflows and that abcd is written beyond the end of the buffer

27 Buffer Overflow: An Example

Summery (cont)
• SoftIce also shows us that the only register with anything interesting in it is ESI
• ESI points to our buffer
• So, ideally, we want a snippet of code loaded into memory that performs a “call ESI” or “jump ESI”
• We simply insert our disassembled code (or any code we want for that matter) into our buffer at the correct spot and fire it off at the open port again

28 Buffer Overflow: An Example

Summery (cont)
• We now have code executing in the remote process!
• That’s it! That is the general way to overflow a buffer!

29 Postmortem Countermeasures

Uuuummmmm...
• There aren’t any, really...
• It takes an elite hacker to perform these types of attacks
• But, they are difficult to discover and difficult to trace
• How do you protect yourself?
• The issue here is that the problem resides with commercial software which is out of our control

30 Postmortem Countermeasures

But, here goes...
• Stay current with patches & updates
• You could install some intrusion detection and prevention software, i.e., Entercept

31 Summary
An elite hacker can locate and exploit buffer overflow conditions.
Then, use that to compromise the system.
The message of this chapter: even with all the security controls, procedures, patches and fixes, you are still vulnerable!