CS 356 Systems Security
Spring 2015
http://www.cs.colostate.edu/~cs356

Dr. Indrajit Ray

http://www.cs.colostate.edu/~indrajit

Original slides by Lawrie Brown. Adapted for CS 356 by Indrajit Ray. Thanks to Professor Mark Stamp, San Jose State University for some of the slides

Chapter 10

Buffer and Stack Overflows

Buffer Overflow

- A very common attack mechanism
  - From 1988 Morris Worm to Code Red, Slammer, Sasser and many others
- Prevention techniques known
- Still a major concern due to
  - Legacy of widely deployed buggy software
  - Continued careless programming techniques

Buffer Overflow Basics

- Caused by programming error
- Allows more data to be stored in a fixed sized buffer than capacity available
  - Buffer can be on stack, heap, global data
- Overwriting adjacent memory locations
  - Corruption of program data
  - Unexpected transfer of control
  - Memory access violation
  - Execution of code chosen by attacker

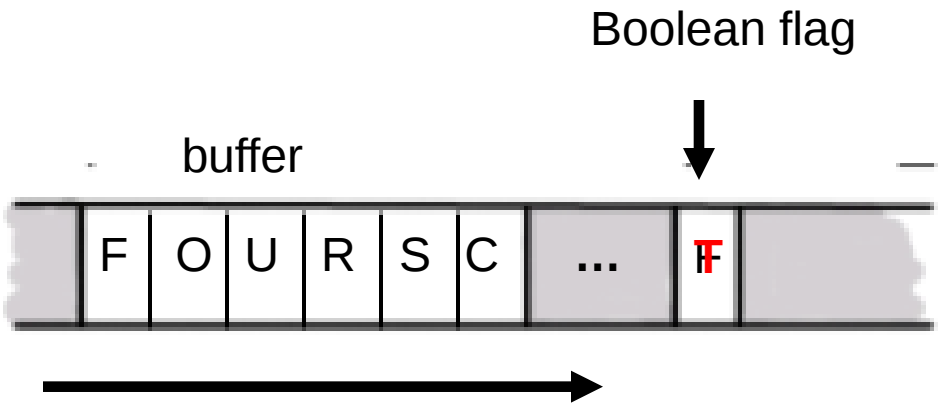
Buffer Overflow

```
int main(){
    int buffer[10];
    buffer[20] = 37;}
```

- **Q:** What happens when this is executed?
- **A:** Depending on what resides in memory at location "buffer[20]"
  - Might overwrite **user** data or code
  - Might overwrite **system** data or code
  - Or program could work just fine

Simple Buffer Overflow

- Consider boolean flag for authentication
- Buffer overflow could overwrite flag allowing anyone to authenticate!

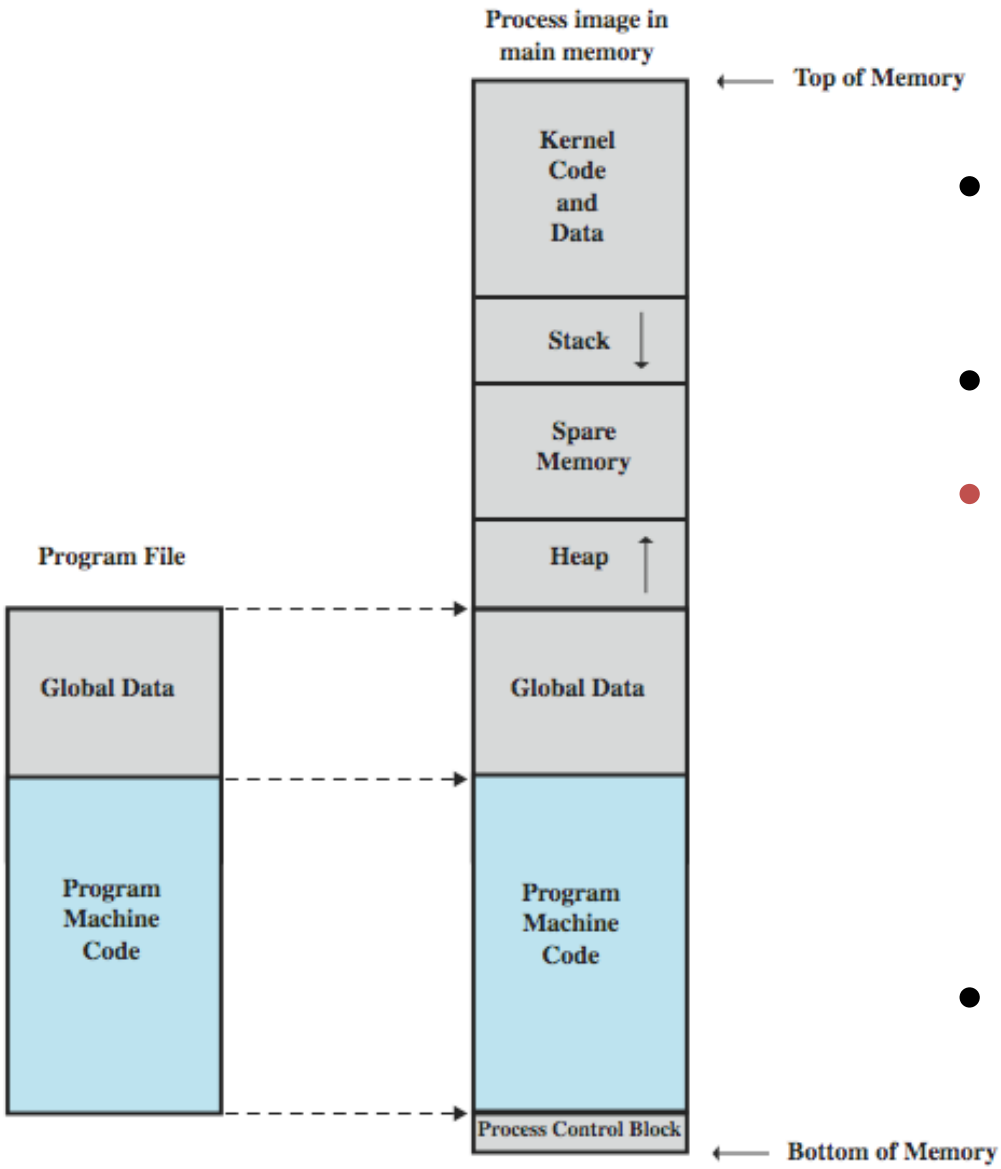


- In some cases, Charlie may not be so lucky as in this example

A Little Programming Language History

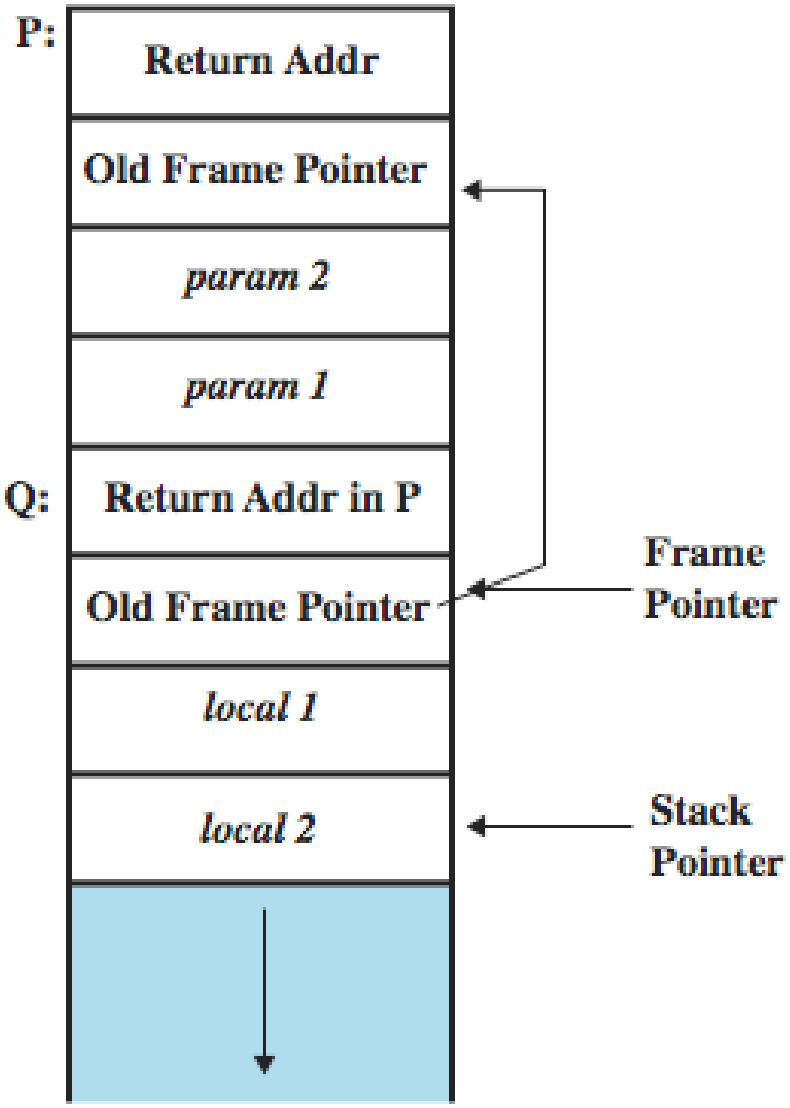
- At machine level all data are array of bytes
  - Interpretation depends on instructions used
- Modern high-level languages have a strong notion of type and valid operations depending on data type
  - Not vulnerable to buffer overflows
  - Does incur overhead, some limits on use
- C and related languages have high-level control structures, but allow direct access to memory
  - Hence are vulnerable to buffer overflow
  - Have a large legacy of widely used, unsafe, and hence vulnerable code

Programs and Processes



- Top of memory = high address
- Data = static variables
- **Stack** == "scratch paper"
  - Dynamic local variables
  - Parameters to functions
  - Return address
- Heap == dynamic data

Function Calls and Stack Frames



Buffer Overflow Attacks

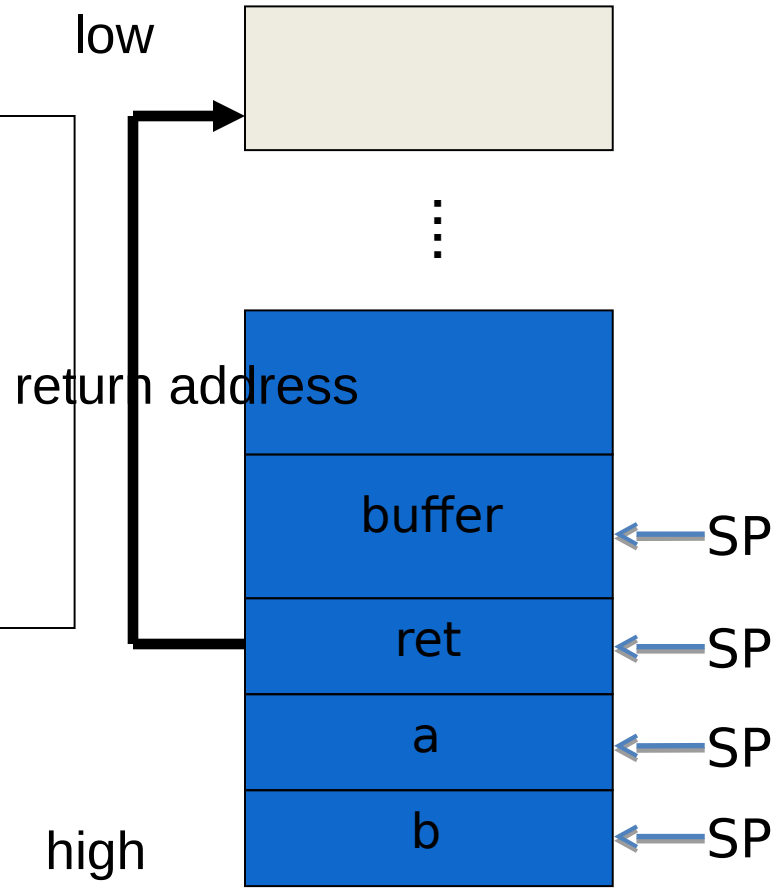
- To exploit a buffer overflow, an attacker
  - Must identify a buffer overflow vulnerability in some program
    - inspection, tracing execution, fuzzing tools
  - Understand how buffer is stored in memory and determine potential for corruption

Stack Buffer Overflow

- Occurs when targeted buffer is located on stack
  - Used by Morris Worm
  - “Smashing the Stack” paper popularized it
- Have local variables below saved frame pointer and return address
  - Hence overflow of a local buffer can potentially overwrite these key control items
- Attacker overwrites return address with address of desired code
  - Program, system library or loaded in buffer

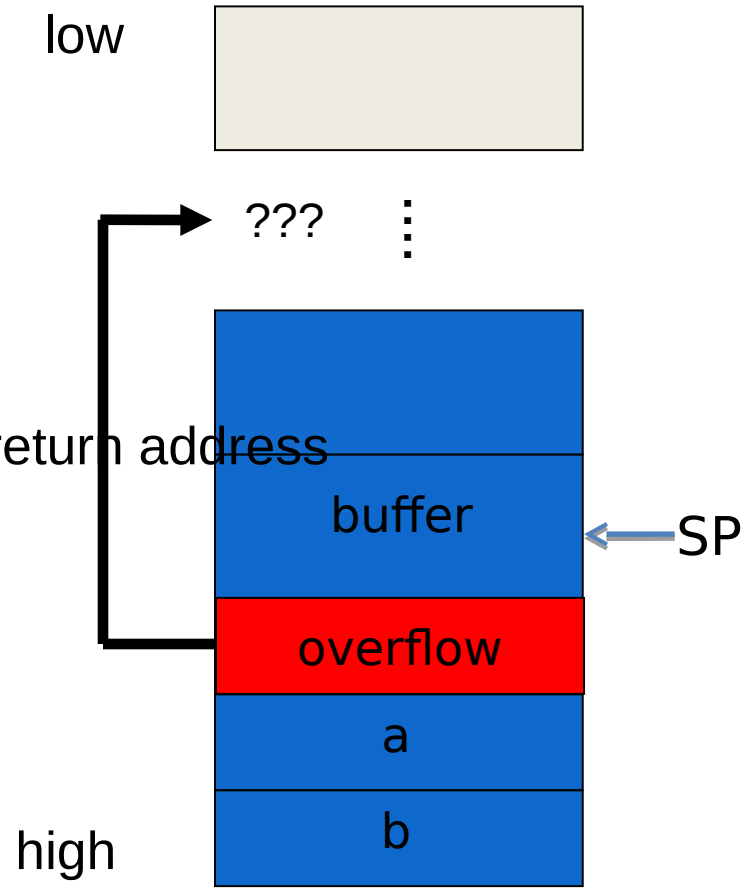
Simplified Stack Example

```
void func(int a, int b){
   char buffer[10];
}
void main(){
   func(1, 2);
}
```



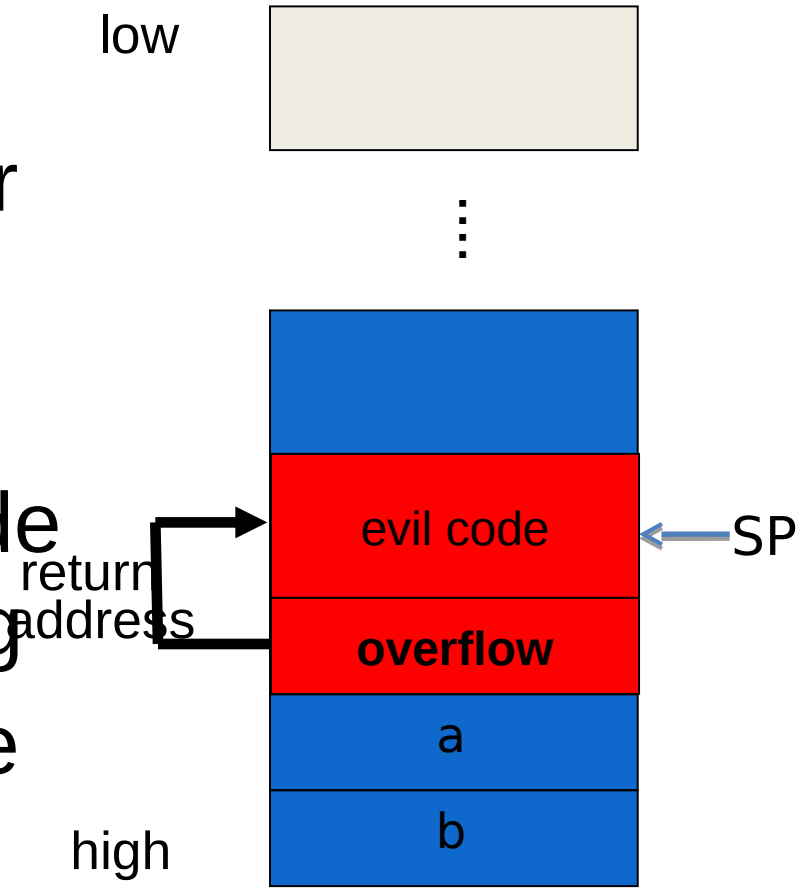
Smashing the Stack

- What happens if buffer overflows?
- Program “returns” to wrong location
- A crash is likely



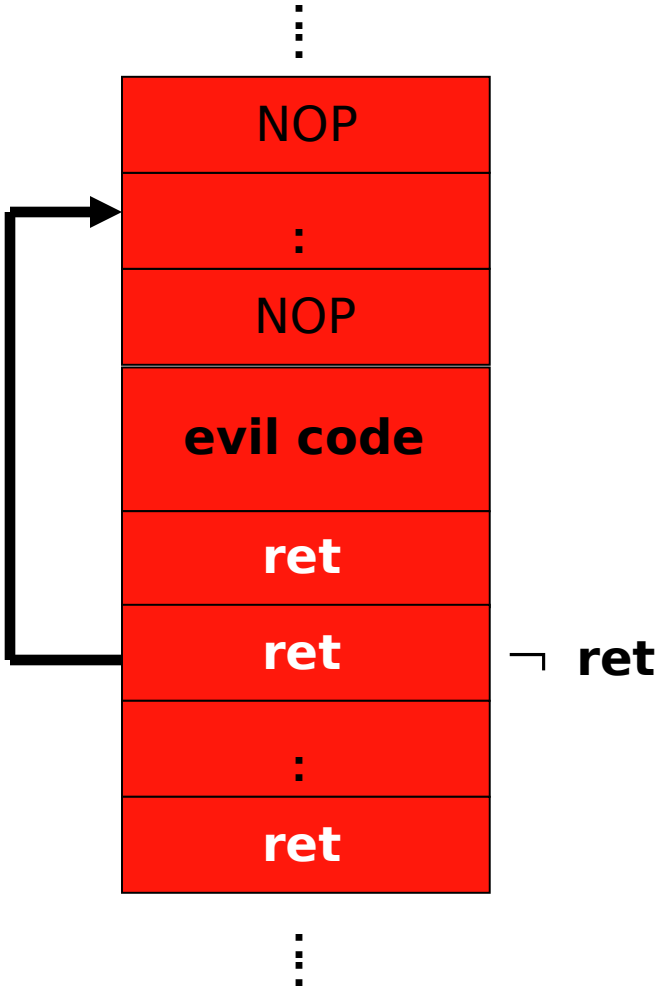
Smashing the Stack

- Charlie has a better idea
- Code injection
- Charlie can run code of his own choosing
  - On your machine



Smashing the Stack

- Charlie may not know…
  1) Address of evil code
  2) Location of **ret** on stack
- Solutions
  1) Precede evil code with NOP "landing pad"
  2) Insert **ret** many times

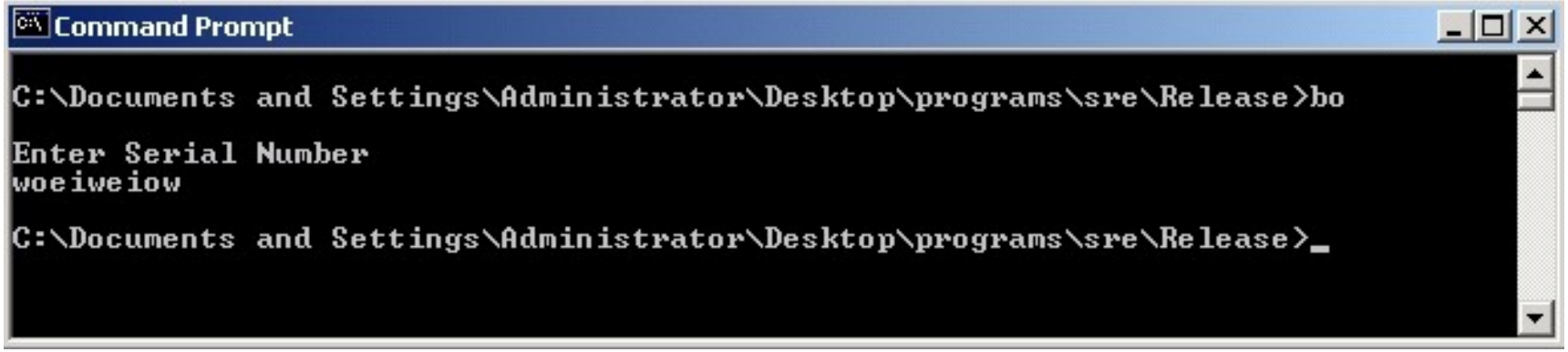


Stack Smashing Summary

- A buffer overflow must exist in the code
- Not all buffer overflows are exploitable
  - Things must align just right
- If exploitable, attacker can **inject code**
- Trial and error is likely required
  - Smashing the Stack for Fun and Profit, Aleph One
- Stack smashing is "attack of the decade"
  - Regardless of the decade…

Stack Smashing Example

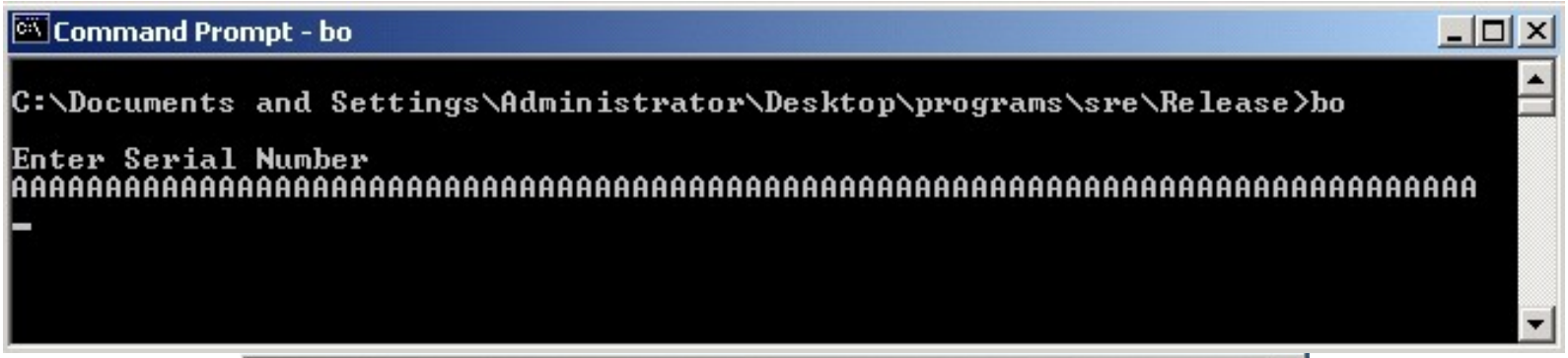
- Program asks for a serial number that the attacker does not know
- Attacker does **not** have source code
- Attacker does have the executable (exe)

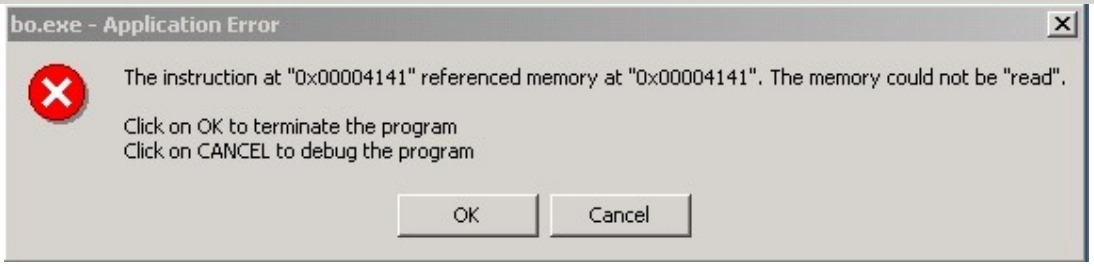


- Program quits on incorrect serial number

Example

- By trial and error, attacker discovers apparent buffer overflow





- Note that 0x41 is “A”
- Looks like **ret** overwritten by 2 bytes!

Example

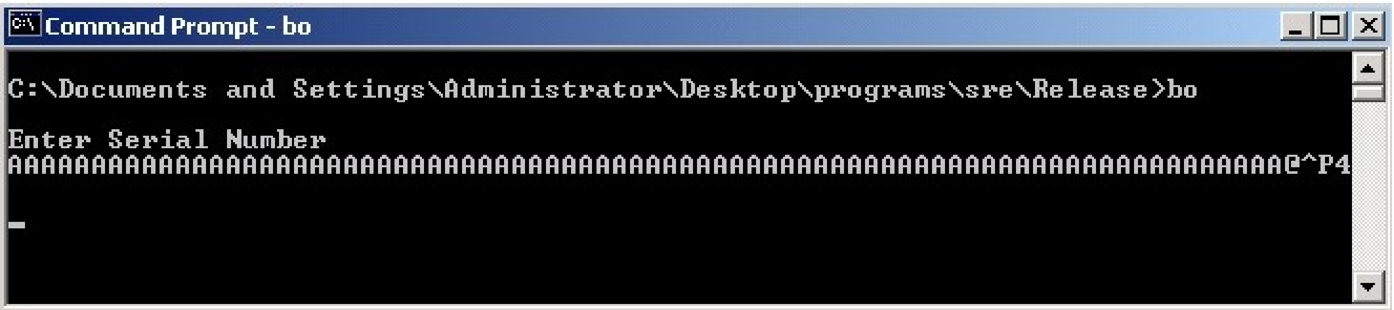
- Next, disassemble bo.exe to find

```
.text:00401000
.text:00401000          sub     esp, 1Ch
.text:00401003          push    offset aEnterSerialNum ; "\nEnter Serial Number\n"
.text:00401008          call    sub_40109F
.text:0040100D          lea     eax, [esp+20h+var_1C]
.text:00401011          push    eax
.text:00401012          push    offset aS       ; "%s"
.text:00401017          call    sub_401088
.text:0040101C          push    8
.text:0040101E          lea     ecx, [esp+2Ch+var_1C]
.text:00401022          push    offset aS123n456 ; "S123N456"
.text:00401027          push    ecx
.text:00401028          call    sub_401050
.text:0040102D          add     esp, 18h
.text:00401030          test    eax, eax
.text:00401032          jnz     short loc_401041
.text:00401034          push    offset aSerialNumberIs ; "Serial number is correct.\n"
.text:00401039          call    sub_40109F
.text:0040103E          add     esp, 4
```

- The goal is to exploit buffer overflow to jump to address 0x401034

Example

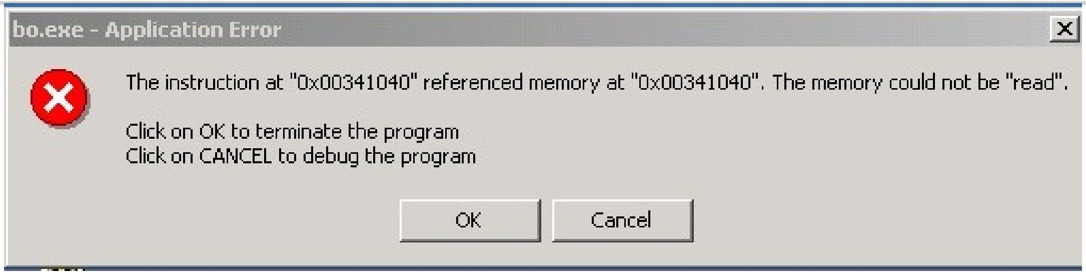
- Find that, in ASCII, 0x401034 is “@^P4”



```
C:\Documents and Settings\Administrator\Desktop\programs\sre\Release>bo

Enter Serial Number
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA@^P4

_
```



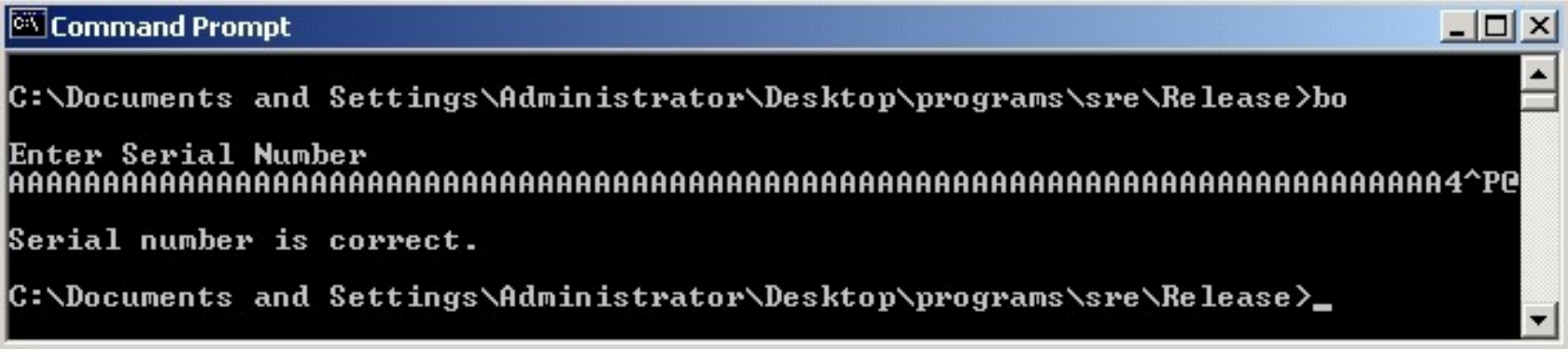
The instruction at "0x00341040" referenced memory at "0x00341040". The memory could not be "read".

Click on OK to terminate the program
Click on CANCEL to debug the program

- ❑ Byte order is reversed? Why?
- ❑ X86 processors are “little-endian”

Example

- Reverse the byte order to “4^P@” and…



- Success! We’ve bypassed serial number check by exploiting a buffer overflow
- What just happened?
  - We overwrote the return address on the stack

Example

- Attacker did **not** require access to the source code
- Only tool used was a disassembler to determine address to jump to
- Find desired address by trial and error?
  - Necessary if attacker does not have exe
  - For example, in a remote attack

Example

- Source code of the buffer overflow
- Flaw easily found by attacker

```
#include <stdio.h>
#include <string.h>

main()
{
    char in[75];

    printf("\nEnter Serial Number\n");

    scanf("%s", in);

    if(!strncmp(in, "S123N456", 8))
    {
        printf("Serial number is correct.\n");
    }
}
```

More Stack Overflow Variants

- Target program can be:
  - A trusted system utility
  - Network service daemon
  - Commonly used library code, e.g. image

Shellcode

- Code supplied by attacker
  - often saved in buffer being overflowed
  - traditionally transferred control to a shell
- Machine code
  - specific to processor and operating system
  - traditionally needed good assembly language skills to create
  - more recently have automated sites/tools

Shellcode

- Shellcode functions
  - Spawn shell
  - Create listener to launch shell on connect
  - Create reverse connection to attacker
  - Flush firewall rules
  - Break out of chroot environment