Attacks on Memory Buffers

- Buffer is a data storage area inside computer memory (stack or heap)
  - Intended to hold pre-defined amount of data
    - If more data is stuffed into it, it spills into adjacent memory
  - If executable code is supplied as “data”, victim’s machine may be fooled into executing it – we’ll see how
    - Code will self-propagate or give attacker control over machine
- First generation exploits: stack smashing
- Second gen: heaps, function pointers, off-by-one
- Third generation: format strings and heap management structures

Stack Buffers

buf

- 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

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

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

Changing Flags

	buf	authenticated	

- Suppose Web server contains this function

```
void func(char *str) {
      int authenticated = 0;
      char buf[126];
      ...
      strcpy(buf,str);
      ...
}
```

- Authenticated variable non-zero when user has extra privileges
- Morris worm also overflowed a buffer to overwrite an authenticated flag in in.fingerd

Changing Flags

buf | authenticated

- Suppose Web server contains this function

```
void func(char *str) {
    int authenticated = 0;
    char buf[126];
    ...
    strcpy(buf,str);
    ...
}
```

- Authenticated variable non-zero when user has extra privileges
- Morris worm also overflowed a buffer to overwrite an authenticated flag in in.fingerd

Changing Flags

- Suppose Web server contains this function

```
void func(char *str) {
    int authenticated = 0;
    char buf[126];
    ...
    strcpy(buf,str);
    ...
}
```

- Authenticated variable non-zero when user has extra privileges
- Morris worm also overflowed a buffer to overwrite an authenticated flag in in.fingerd

Changing Flags

- Suppose Web server contains this function

```
void func(char *str) {
    int authenticated = 0;
    char buf[126];
    ...
    strcpy(buf,str);
    ...
}
```

- Authenticated variable non-zero when user has extra privileges
- Morris worm also overflowed a buffer to overwrite an authenticated flag in in.fingerd

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

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

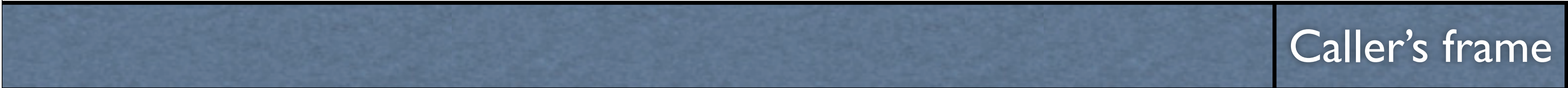
- Suppose Web server contains this function

```
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 with local variables is pushed onto the stack



Addr 0xFF...F

Stack Buffers

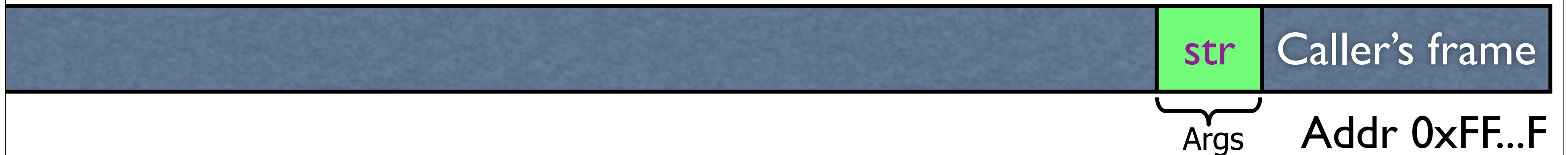
- Suppose Web server contains this function

```
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 with local variables is pushed onto the stack



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 with local variables is pushed onto the stack

Stack Buffers

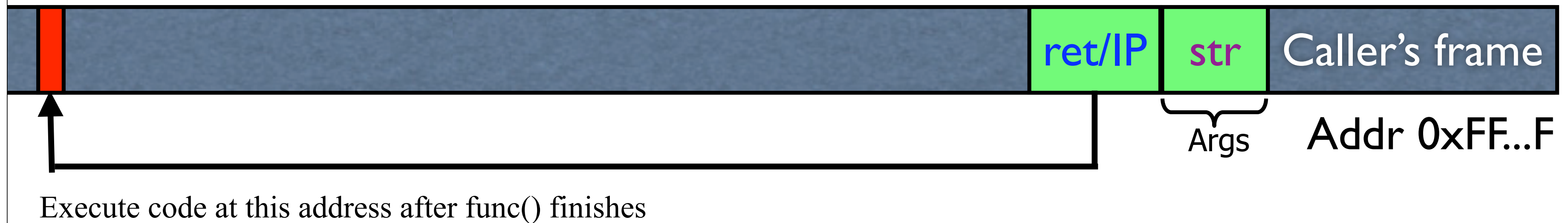
- Suppose Web server contains this function

```
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 with local variables is pushed onto the stack



Execute code at this address after func() finishes

Stack Buffers

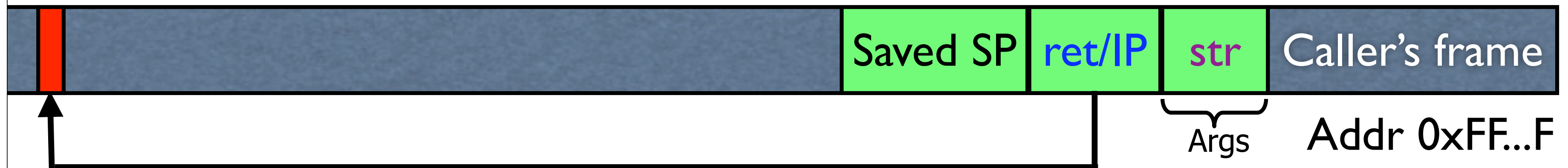
- Suppose Web server contains this function

```
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 with local variables is pushed onto the stack



Execute code at this address after func() finishes

Stack Buffers

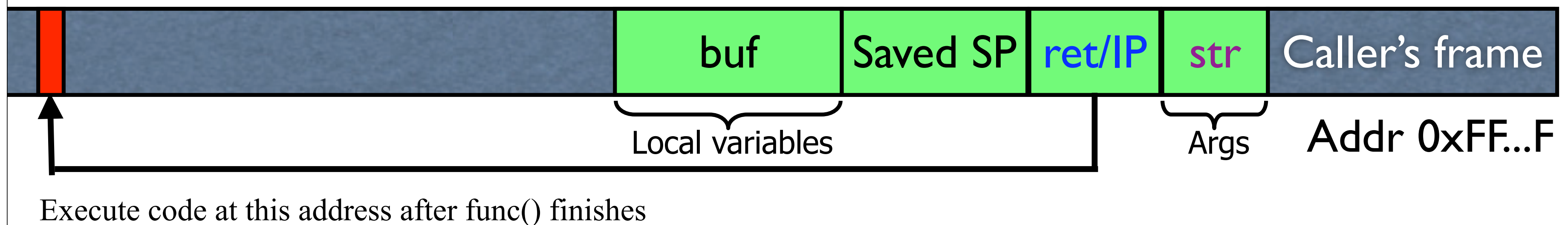
- Suppose Web server contains this function

```
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 with local variables is pushed onto the stack



Execute code at this address after func() finishes

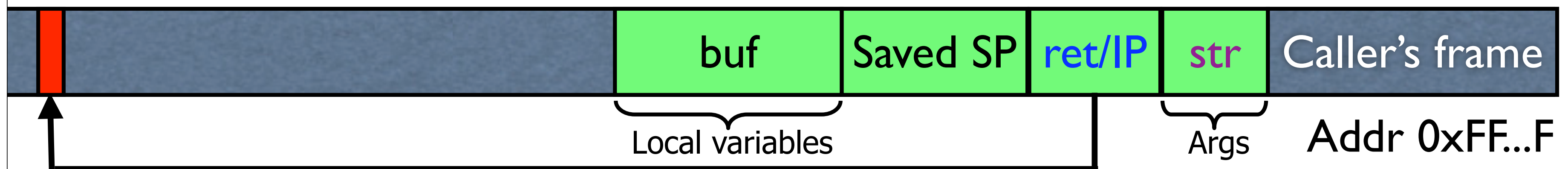
What If Buffer is Overstuffed?

- 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



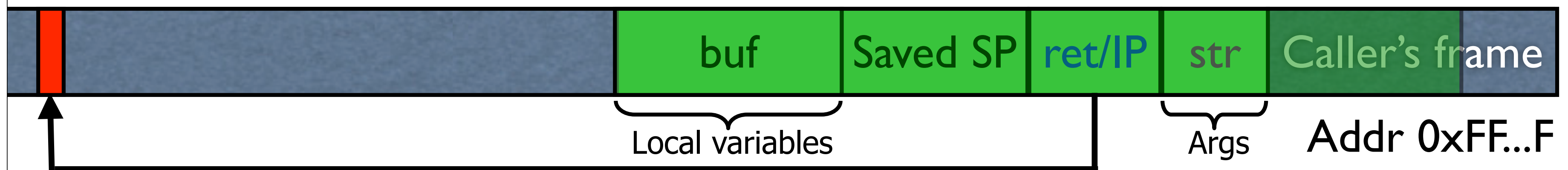
What If Buffer is Overstuffed?

- 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



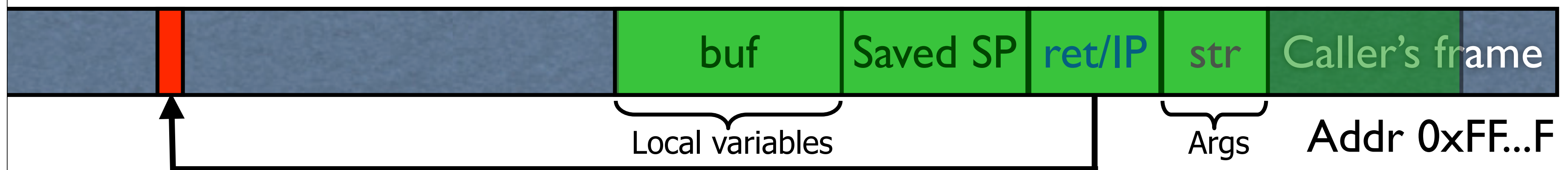
What If Buffer is Overstuffed?

- Memory pointed to by str is copied onto stack…

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

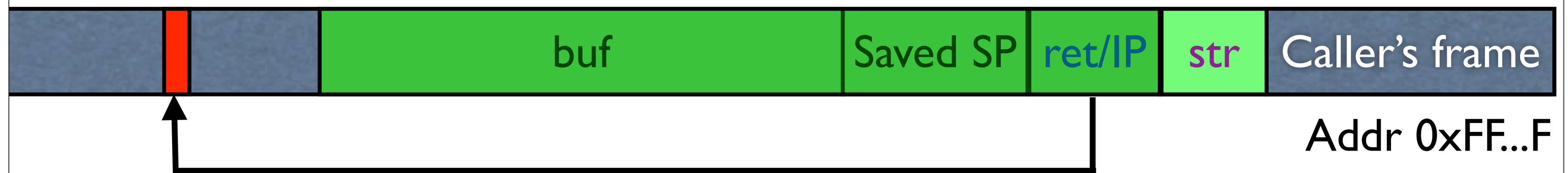
strcpy does NOT check whether the string at *str contains fewer than 126 characters

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



Executing Attack Code

- Suppose buffer contains attacker-created string
  - For example, *str contains a string received from the network as input to some network service daemon



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

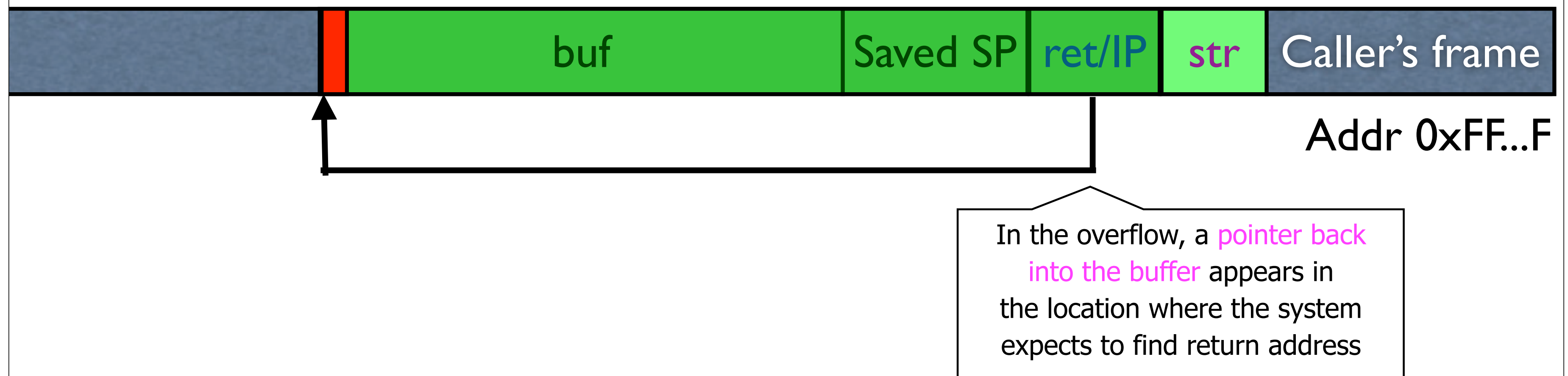
Executing Attack Code

- Suppose buffer contains attacker-created string
  - For example, *str contains a string received from the network as input to some network service daemon

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

Executing Attack Code

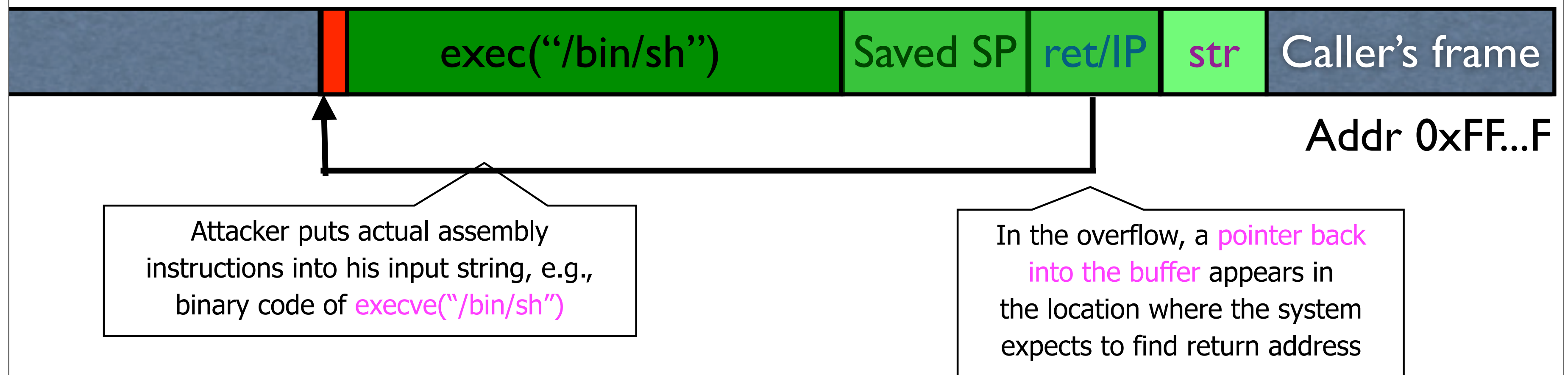
- Suppose buffer contains attacker-created string
  - For example, *str contains a string received from the network as input to some network service daemon



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

Executing Attack Code

- Suppose buffer contains attacker-created string
  - For example, *str contains a string received from the network as input to some network service daemon



- 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 crash with segmentation violation
  - Attacker must correctly guess in which stack position his buffer will be when the function is called

Problem: No Range 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 Range 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" (do you see problem?):

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

Off-By-One Overflow

- Home-brewed range-checking string copy

```
void notSoSafeCopy(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)
          notSoSafeCopy(argv[1]);
}
```

Off-By-One Overflow

- Home-brewed range-checking string copy

```
void notSoSafeCopy(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)
        notSoSafeCopy(argv[1]);
}
```

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

Off-By-One Overflow

- Home-brewed range-checking string copy

```
void notSoSafeCopy(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)
        notSoSafeCopy(argv[1]);
}
```

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

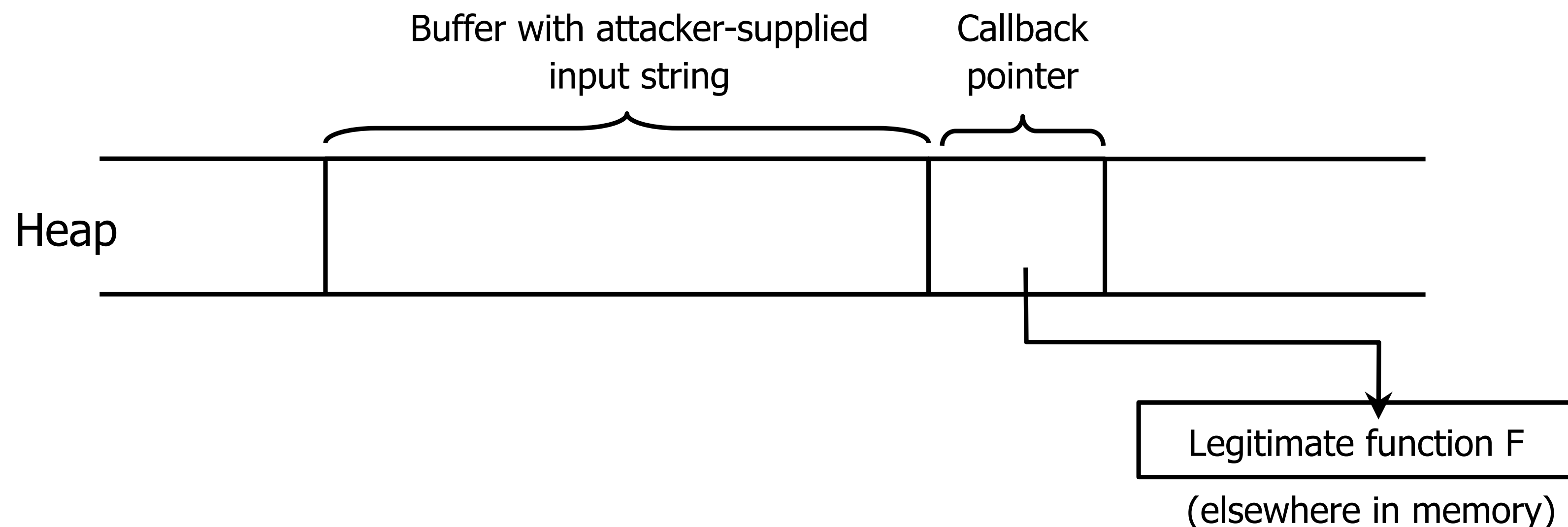
- 1-byte overflow: can't change RET, but can change pointer to previous stack frame
  - On little-endian architecture, make it point into buffer
  - RET for previous function will be read from buffer!

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

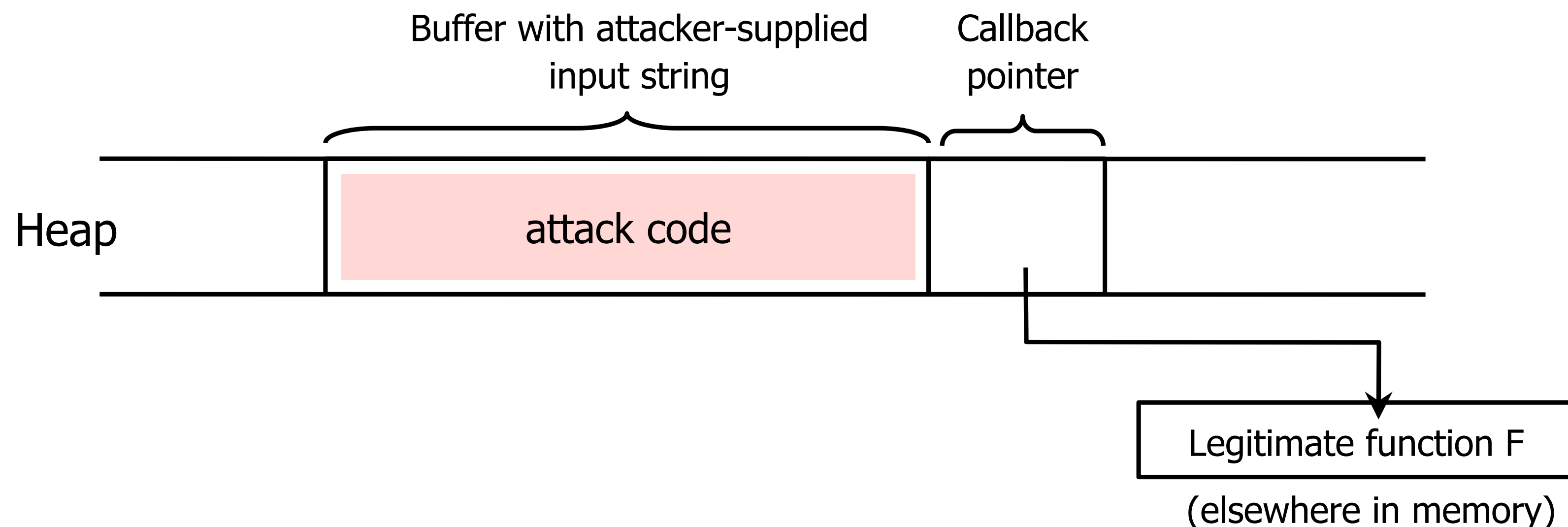
Function Pointer Overflow

- 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)(…)



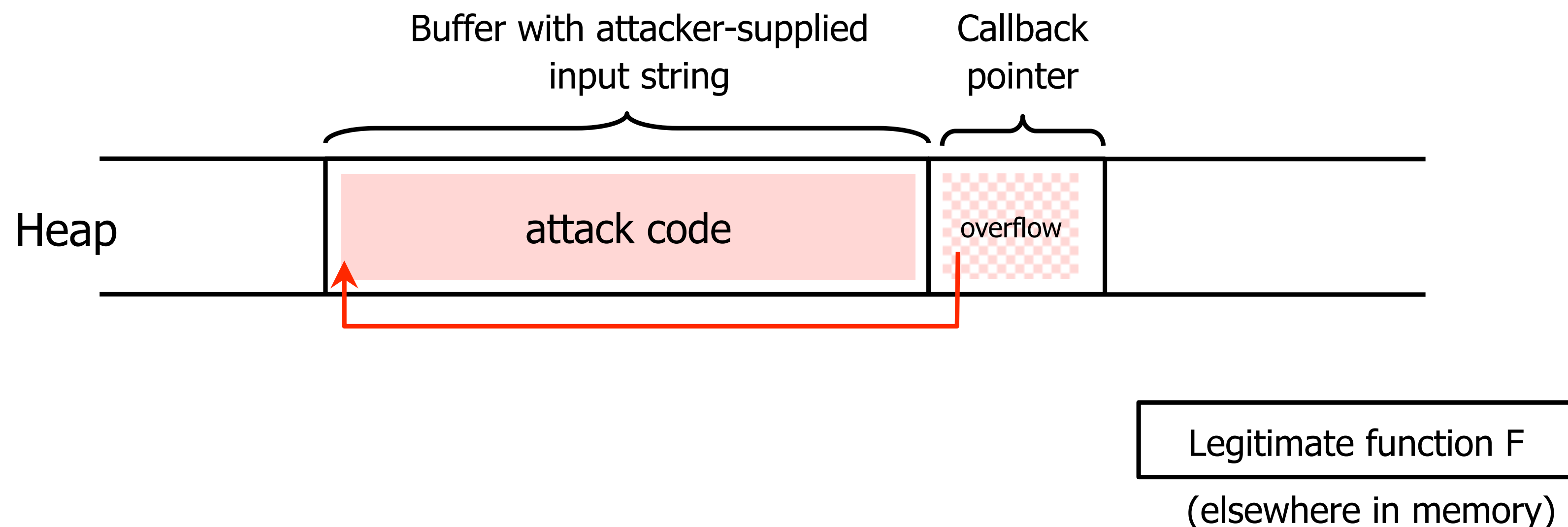
Function Pointer Overflow

- 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)(…)



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)(…)



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); …
```

  – If buffer contains format symbols starting with %, 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.

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.

Lab: GDB will be helpful too!

- disassemble
- run
- continue
- break
  - break main
  - break *0x08048643
- step / stepi
- info register
- x
  - x/200x buf
  - x/200i buf
  - x/200a buf
  - x/200x $sp - 16