Buffer Overflow Attacks

Yan Huang

Credit: Vitaly Shmatikov (Cornell Tech, CS361)

Reading Assignment

- You MUST read Smashing the Stack for Fun and Profit to understand how to start on the project
- Read Once Upon a free()
  - Also on malloc() exploitation: Vudo - An Object Superstitiously Believed to Embody Magical Powers
- Read Exploiting Format String Vulnerabilities

Morris Worm

- 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 a computer science professor at MIT
- Morris claimed it was intended to harmlessly measure the Internet, but it created new copies as fast as it could and overloaded infected hosts
- $10-100M worth of damage

Morris Worm and Buffer Overflow

- We will look at the Morris worm in more detail when talking about worms and viruses
- One of the worm's propagation techniques was a ***buffer overflow*** attack against a vulnerable version of **fingerd** on VAX systems
  - By sending a 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)

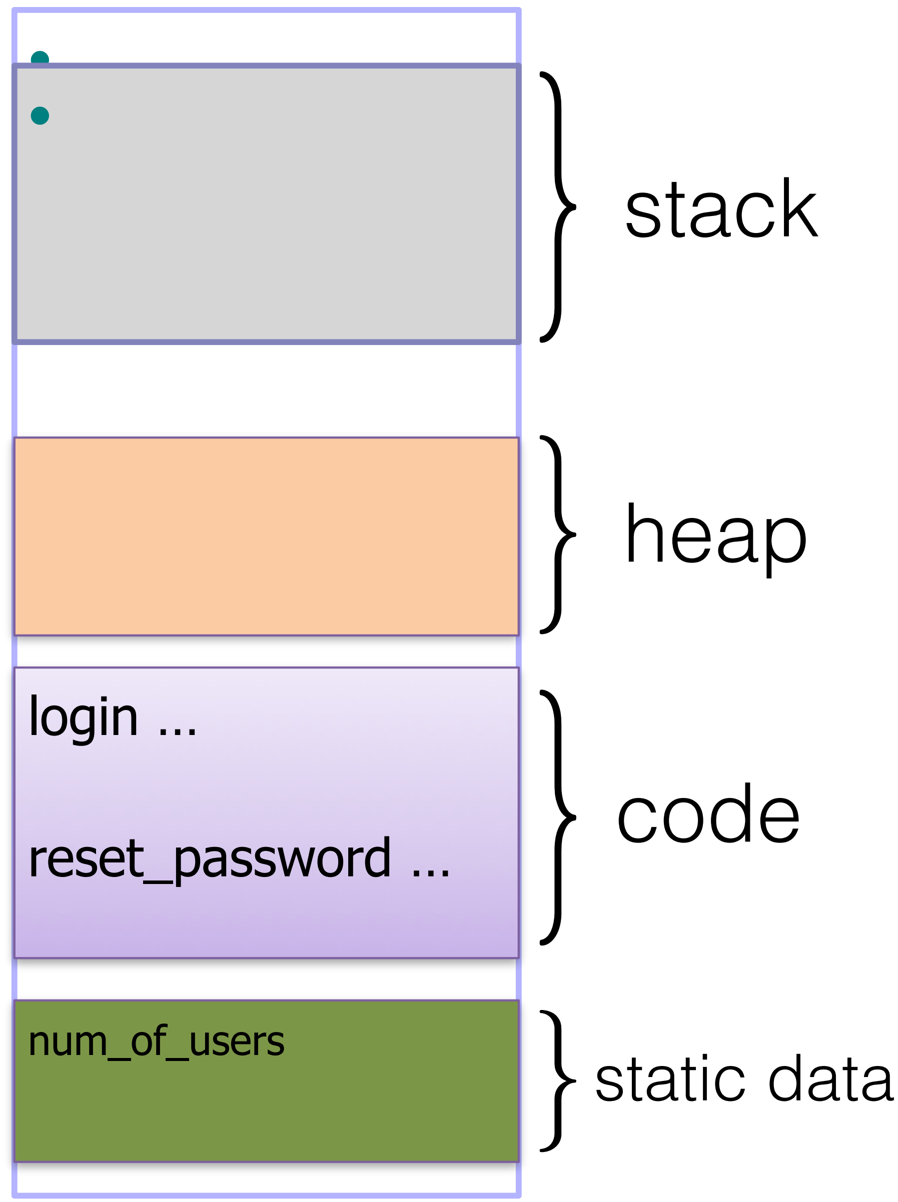
What is buffer overflow?

Memory Layout Review

```
bool num_of_users = 0;

bool login () {
  ……
  if (password_expires())
    reset_password();
  ……
}

void reset_password() {
  ……
  char usr[20], char pwd[100];
  gets(&usr); gets(&pwd);
  update_hash_file(usr,
      compute_hash(pwd, salt));
}
```



Review — What's on the stack?

PC →

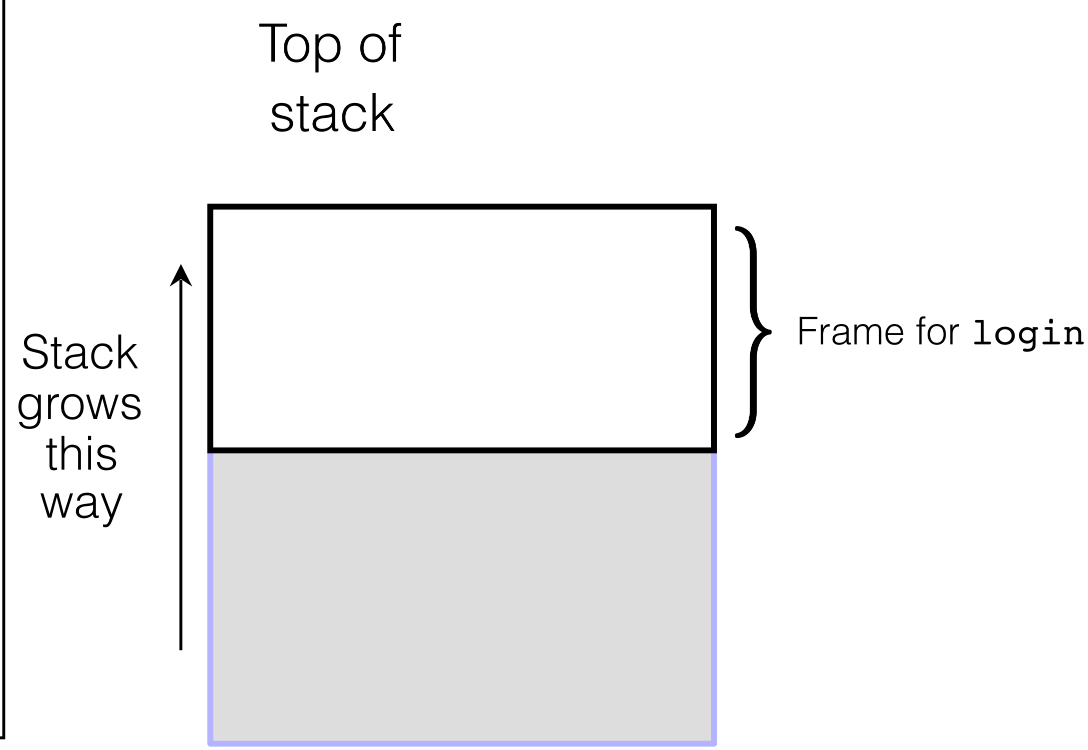
```
bool num_of_users = 0;

bool login () {
  ......
  if (password_expires())
    reset_password();
  ......
}

void reset_password() {
  ......
  char usr[20], char pwd[100];
  gets(&usr); gets(&pwd);
  update_hash_file(usr,
        compute_hash(pwd, salt));
}
```

Each frame has:

- local data for the function
- a pointer to the previous stack frame (**sfp**)
- the value of previous PC (**ret address**)



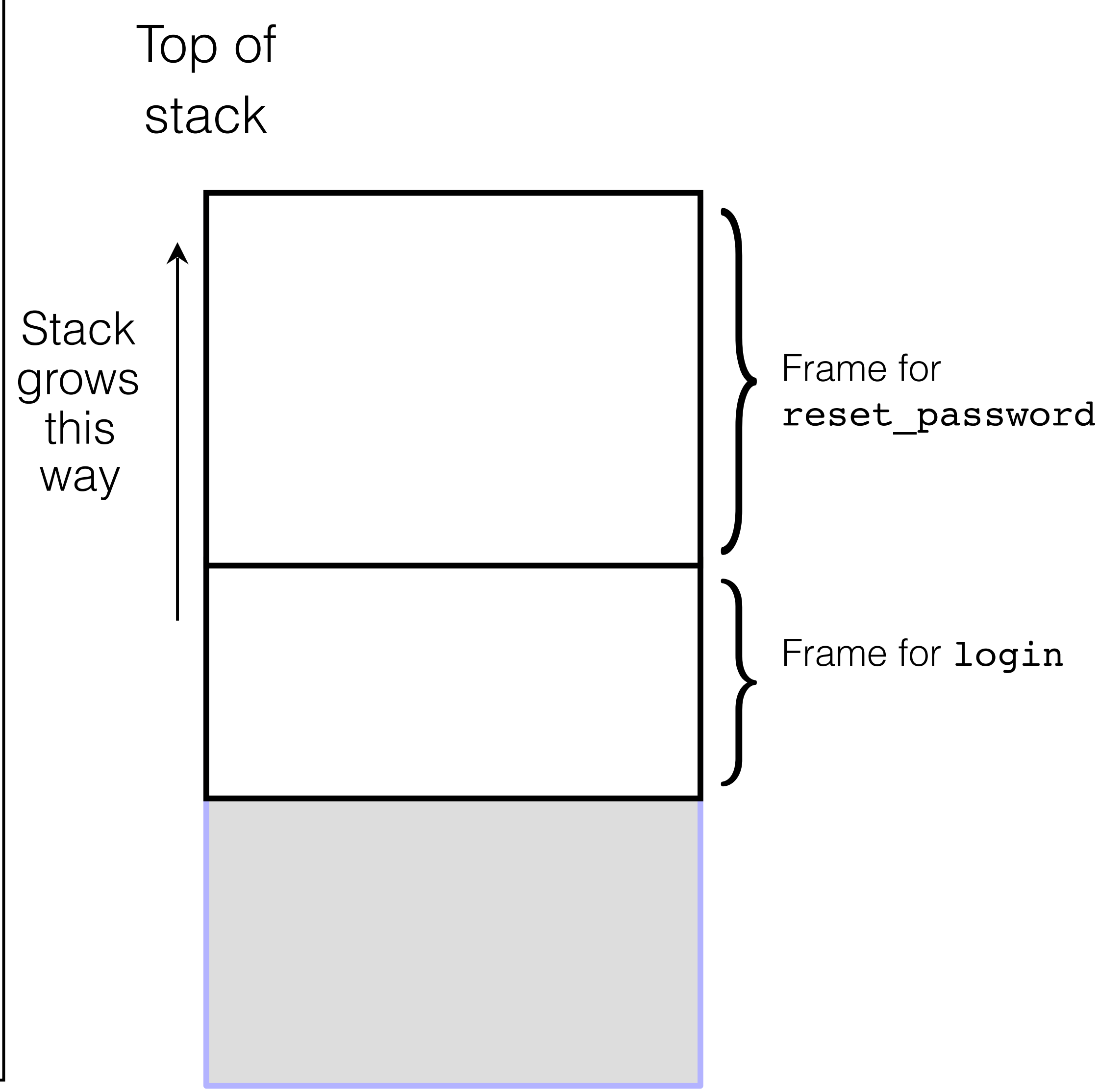
Review — What's on the stack?

```
bool num_of_users = 0;

bool login () {
  ......
  if (password_expires())
    reset_password();
  ......
}

void reset_password() {
  ......
  char usr[20], char pwd[100];
  gets(&usr); gets(&pwd);
  update_hash_file(usr,
        compute_hash(pwd, salt));
}
```

PC



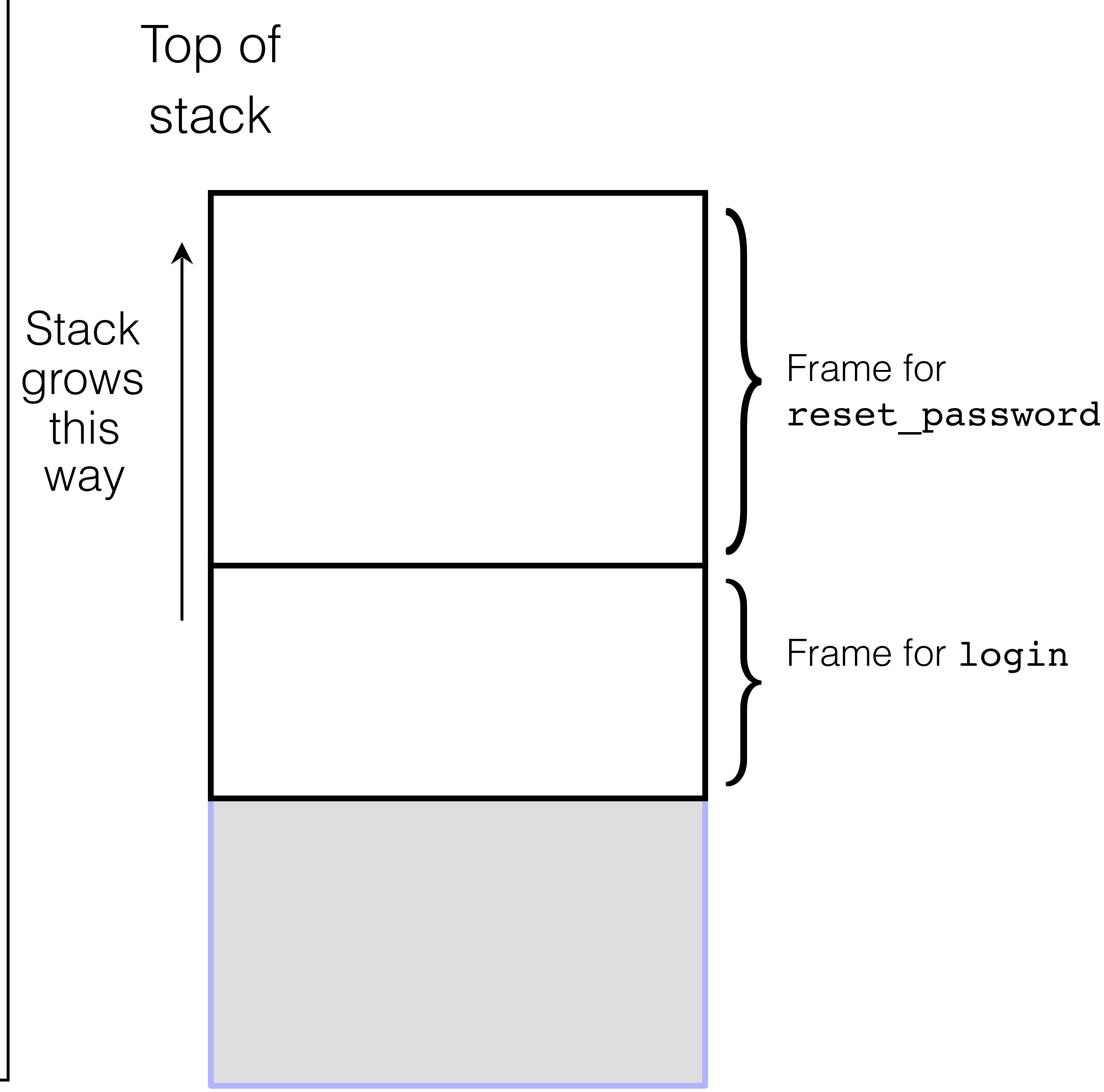
Review — What's on the stack?

```
bool num_of_users = 0;

bool login () {
  ……
  if (password_expires())
    reset_password();
  ……
}

void reset_password() {
  ……
  char usr[20], char pwd[100];
  gets(&usr); gets(&pwd);
  update_hash_file(usr,
      compute_hash(pwd, salt));
}
```

PC →



Review — What's on the stack?

PC →

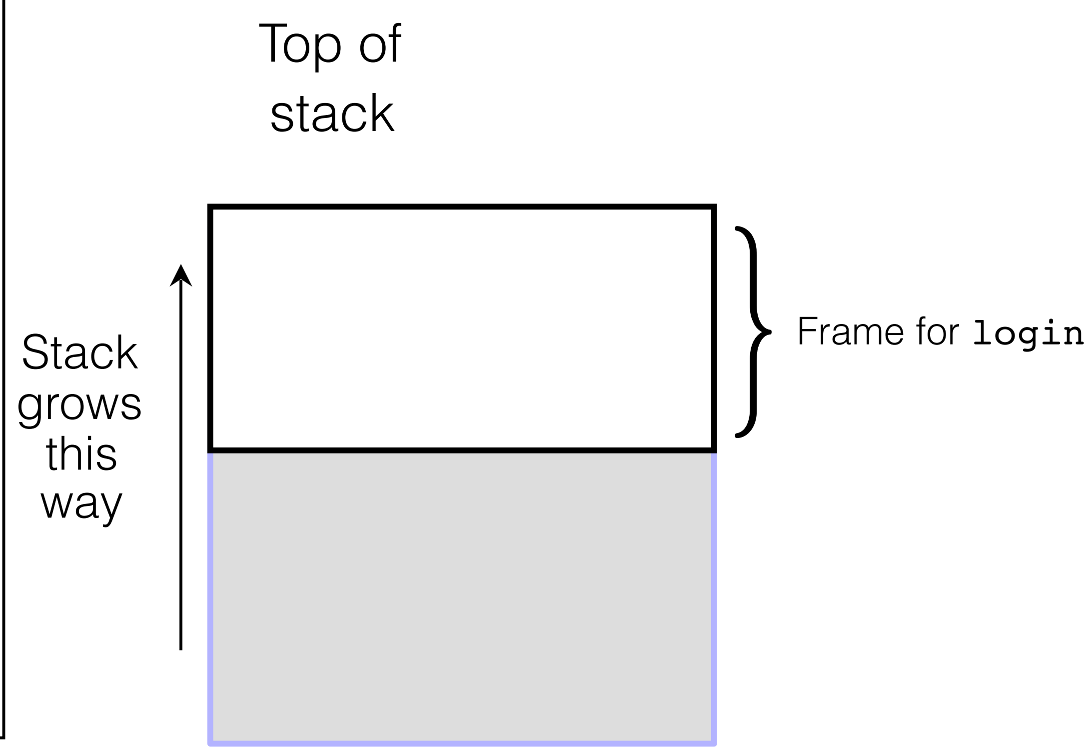
```
bool num_of_users = 0;

bool login () {
  ……
  if (password_expires())
    reset_password();
  ……
}

void reset_password() {
  ……
  char usr[20], char pwd[100];
  gets(&usr); gets(&pwd);
  update_hash_file(usr,
        compute_hash(pwd, salt));
}
```

How can the PC be correctly set upon return?

How does the frame know how much to shrink?



Memory Exploits

- Buffer is a data storage area inside computer memory (stack or heap)
  - Intended to hold pre-defined amount of data
  - If executable code is supplied as “data”, victim’s machine may be fooled into executing it
    - Code will self-propagate or give attacker control over machine
  - Many attacks do not involve executing “data”
- Attack can exploit <u>any</u> memory operation
  - Pointer assignment, format strings, memory allocation and de-allocation, function pointers, calls to library routines via offset tables …

Stack Buffers

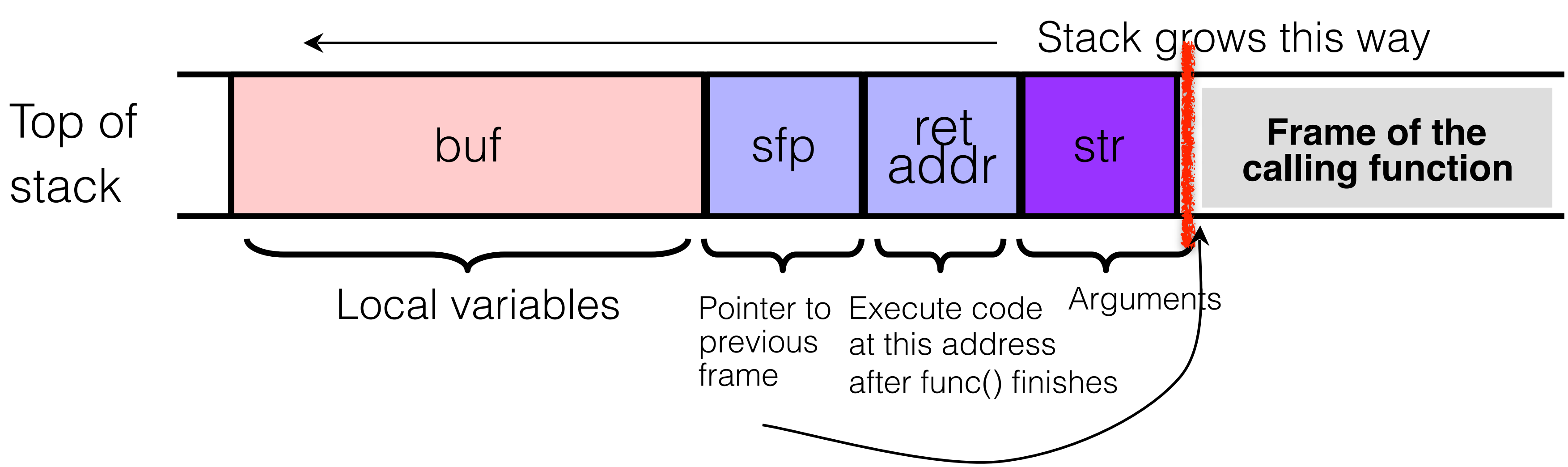
- Suppose Web server contains this function

```
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



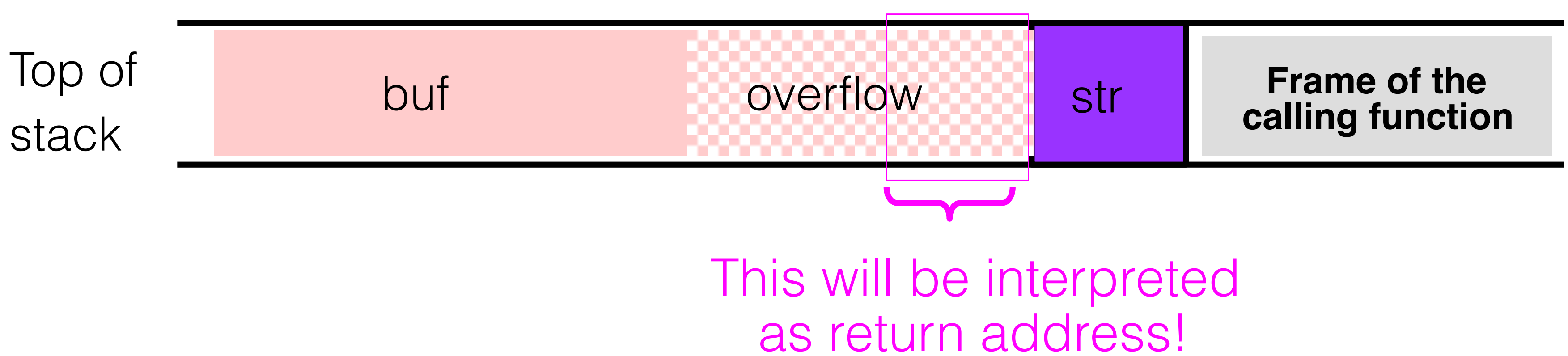
What If Buffer Is Overstuffed?

- Memory pointed to by str is copied onto stack…

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

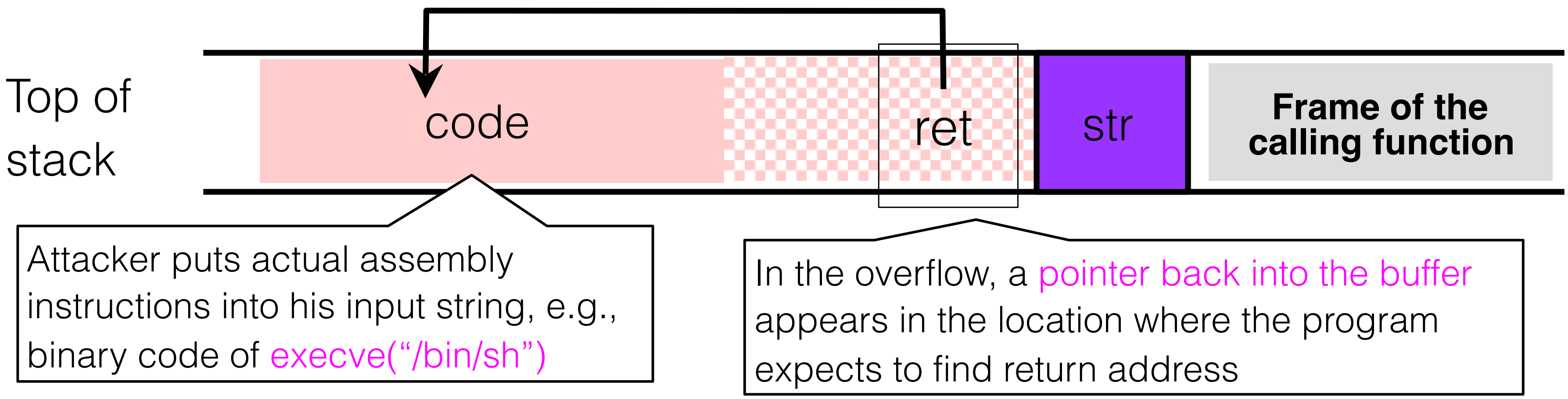
strcpy does NOT check whether the string at *str contains fewer than 126 characters

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



Executing Attack Code

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

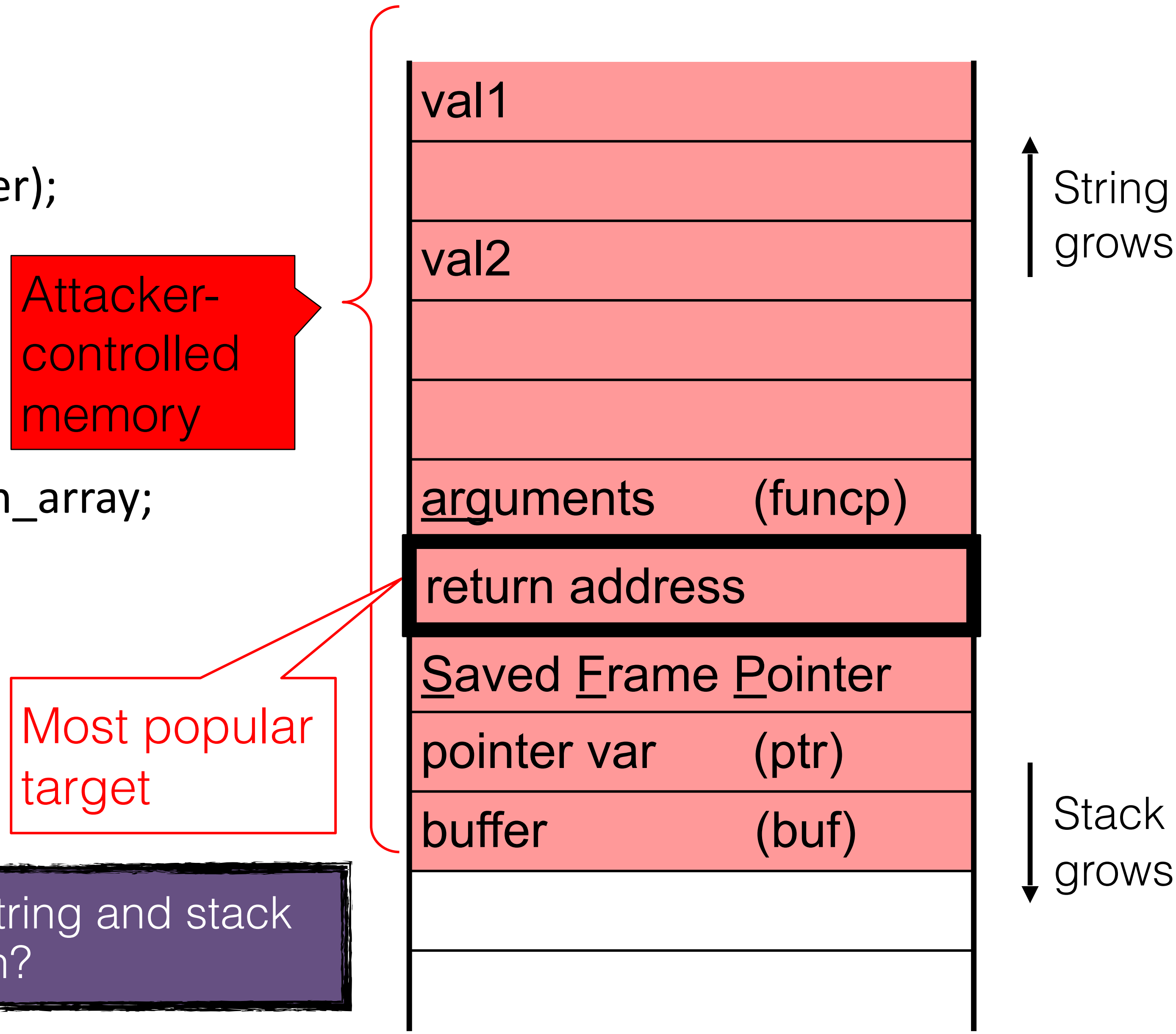


- When function exits, code in the buffer will be executed, giving attacker a shell
  - Root shell if the victim program is setuid root

Stack Corruption: General View

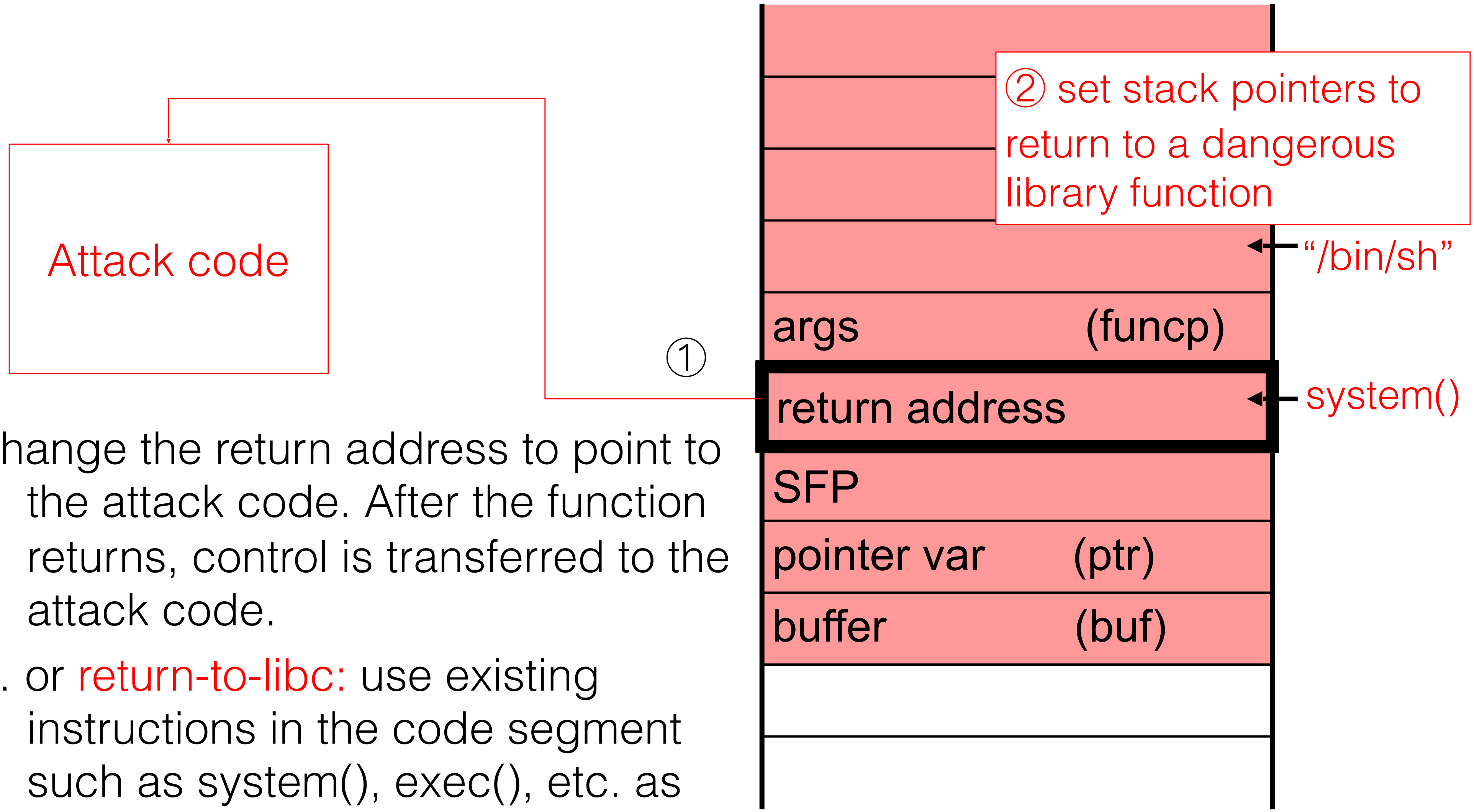
```
int bar (int val1) {
   int  val2;
   foo (a_function_pointer);
}

int foo (void (*funcp)()) {
   char* ptr = point_to_an_array;
   char buf[128];
   gets (buf);
   strncpy(ptr, buf, 8);
   (*funcp)();
}
```



How about dictating the string and stack grow in the same direction?

Attack #1: Return Address



Change the return address to point to the attack code. After the function returns, control is transferred to the attack code.

… or return-to-libc: use existing instructions in the code segment such as system(), exec(), etc. as the attack code.

Basic Stack Code Injection

- Executable attack code is stored on stack, inside the buffer containing attacker's string
  - Stack memory is supposed to contain only data, but…
- For the basic stack-smashing attack, 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 the "attack assembly code" in the buffer
    - Otherwise application will crash with segmentation violation
  - Attacker must correctly guess the position of his stack buffer when the function is called

Cause: No Range Checking

- strcpy does <u>not</u> 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
- Standard C library functions are all unsafe
  - strcpy(char *dest, const char *src)
  - strcat(char *dest, const char *src)
  - gets(char *s)
  - scanf(const char *format, …)
  - printf(const char *format, …)

Did 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 the problem?):

```
… 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); …
```