Buffer Overflow and Stack Smashing

Vitaly Shmatikov

Reading Assignment

- "Smashing the Stack for Fun and Profit" by Aleph One
  - Linked from the course website
- Homework 2 can be done in 2-student teams

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 a computer science 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 Buffer Overflow Attacks

- Morris worm (1988): overflow in fingerd
  - 6,000 machines infected (10% of existing Internet)
- 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 (2004): overflow in Windows LSASS
  - Around 500,000 machines infected

Responsible for user authentication in Windows

- Conficker (2008-09): overflow in Windows Server
  - Around 10 million machines infected (estimates vary)

Why Are We Insecure?

[Chen et al. 2005]

- 126 CERT security advisories (2000-2004)
- Of these, 87 are memory corruption vulnerabilities
- 73 are in applications providing remote services
  - 13 in HTTP servers, 7 in database services, 6 in remote login services, 4 in mail services, 3 in FTP services
- Most exploits involve illegitimate control transfers
  - Jumps to injected attack code, return-to-libc, etc.
  - Therefore, most defenses focus on control-flow security
- But exploits can also target configurations, user data and decision-making values

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
- Attack can exploit any memory operation
  - Pointer assignment, format strings, memory allocation and de-allocation, function pointers, calls to library routines via offset tables

Stack Buffers

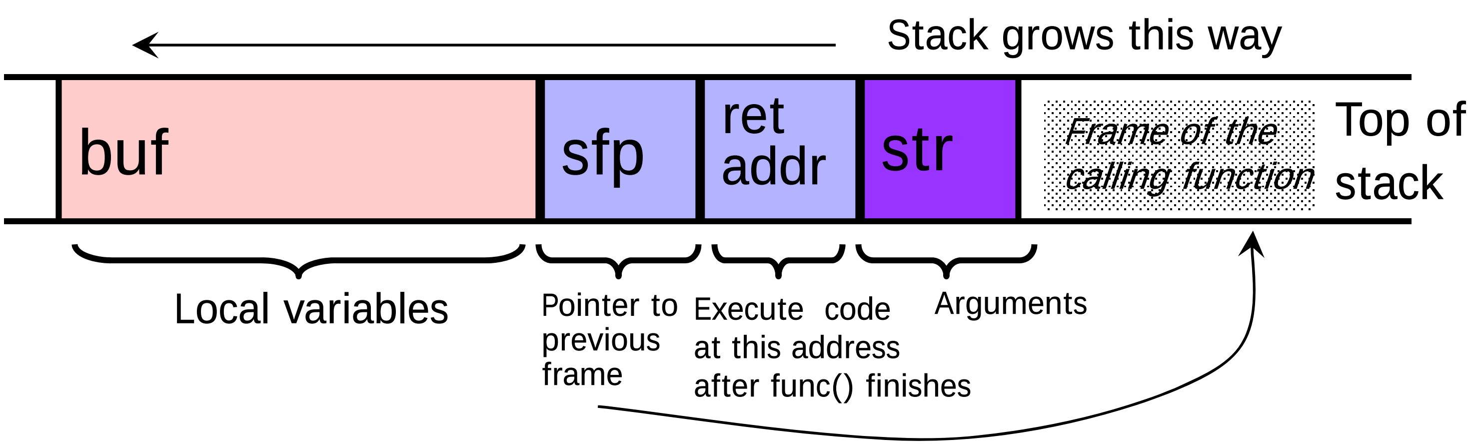
- Suppose Web server contains this function

```
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 with local variables is pushed onto the stack



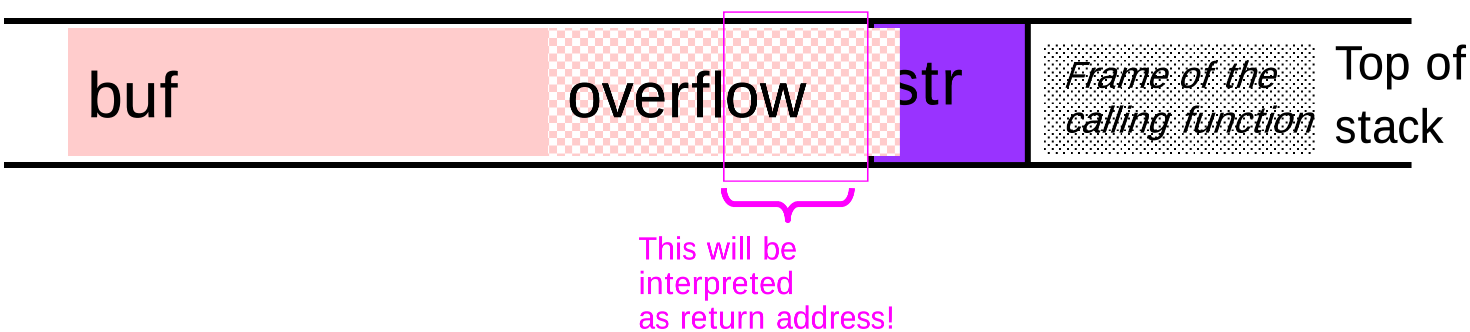
What If Buffer is Overstuffed?

- Memory pointed to by str is copied onto stack...

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

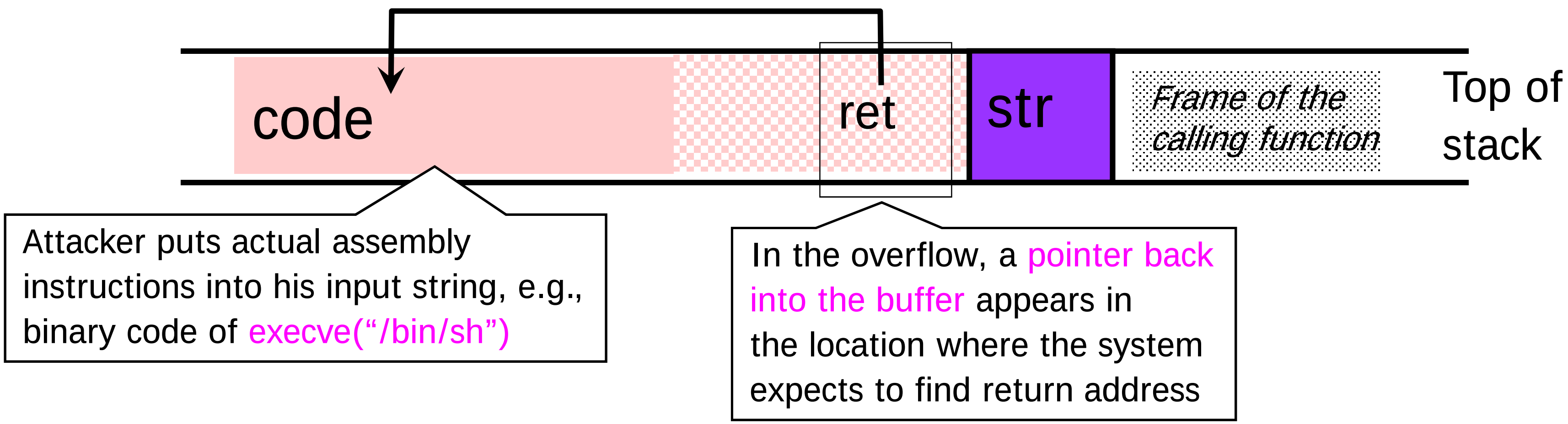
strcpy does NOT check whether the string at *str contains fewer than 126 characters

- 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 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

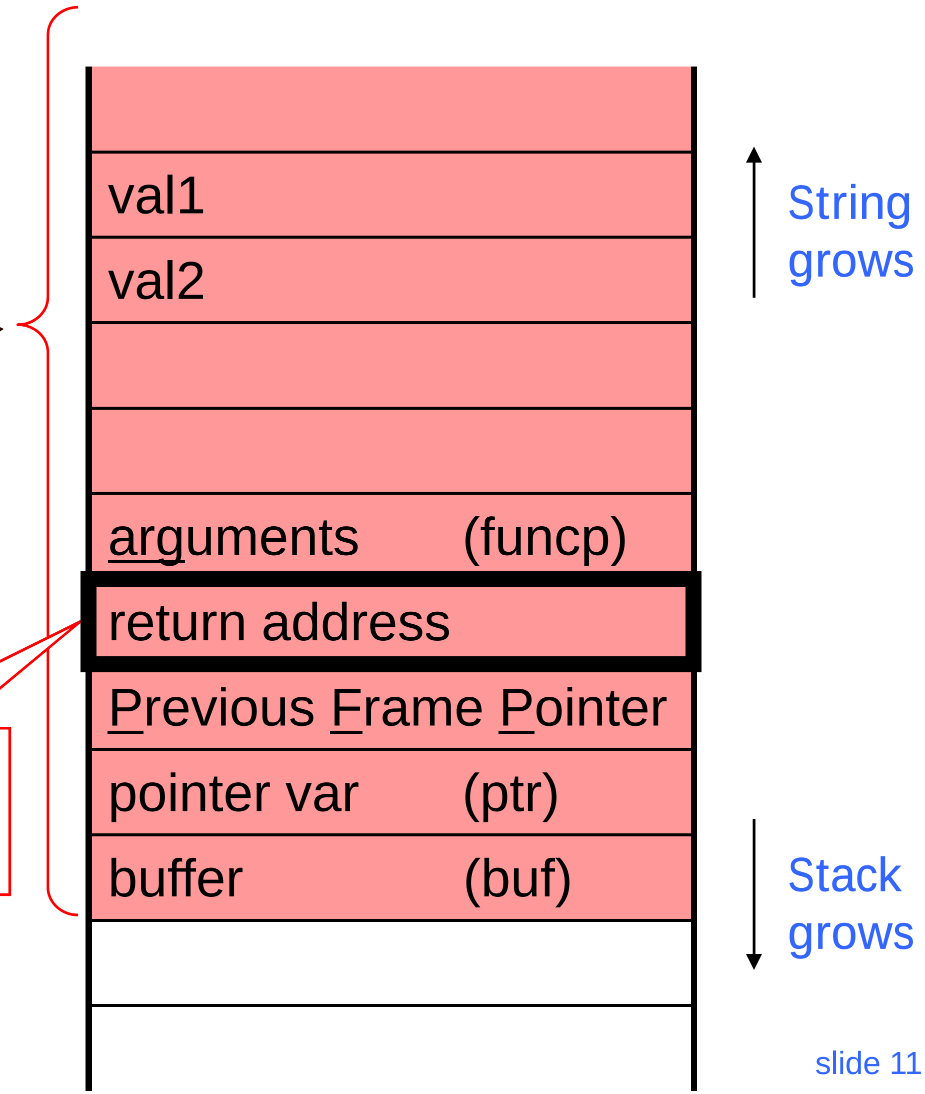
Stack Corruption (Redux)

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

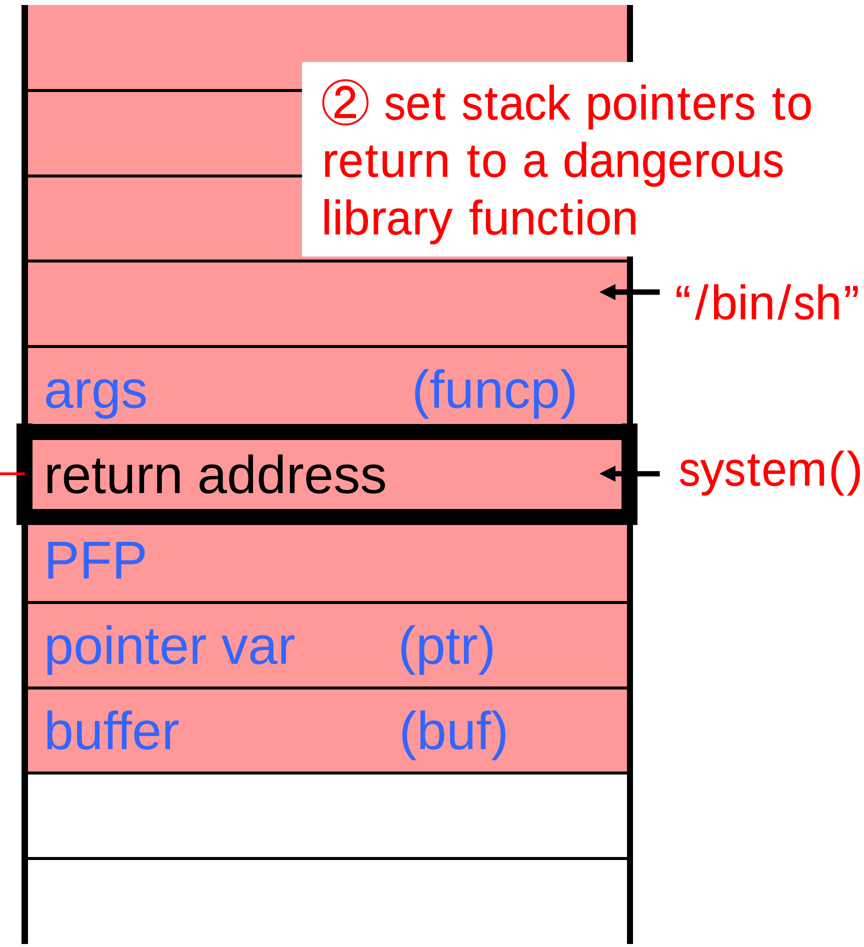
Contaminated memory

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

Most popular target



Attack #1: Return Address



① Change the return address to point to the attack code. After the function returns, control is transferred to the attack code

② ... or return-to-libc: use existing instructions in the code segment such as system(), exec(), etc. as the attack code

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...
- For the basic 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 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")

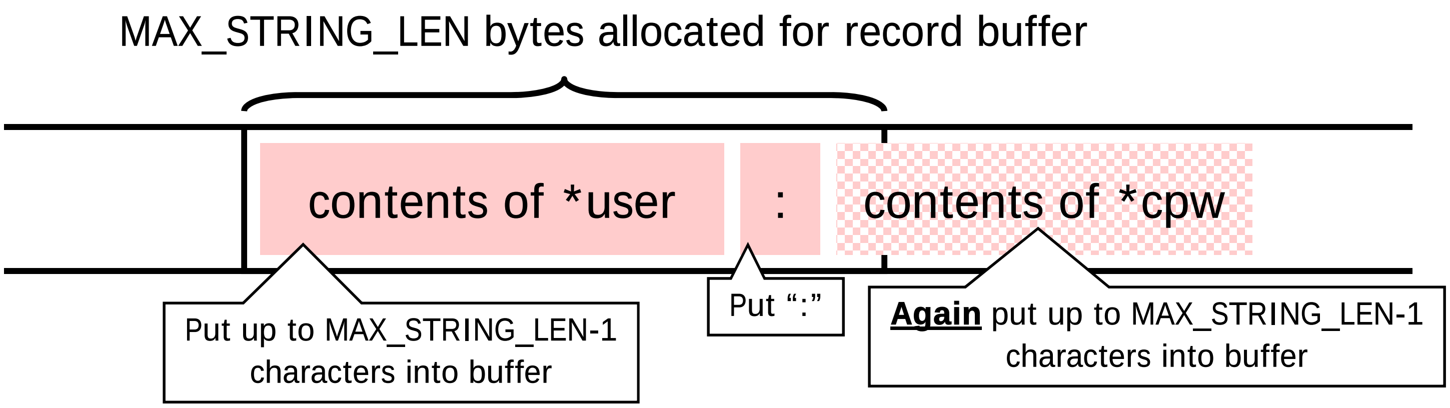
- Published "fix" (do you see the problem?):

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

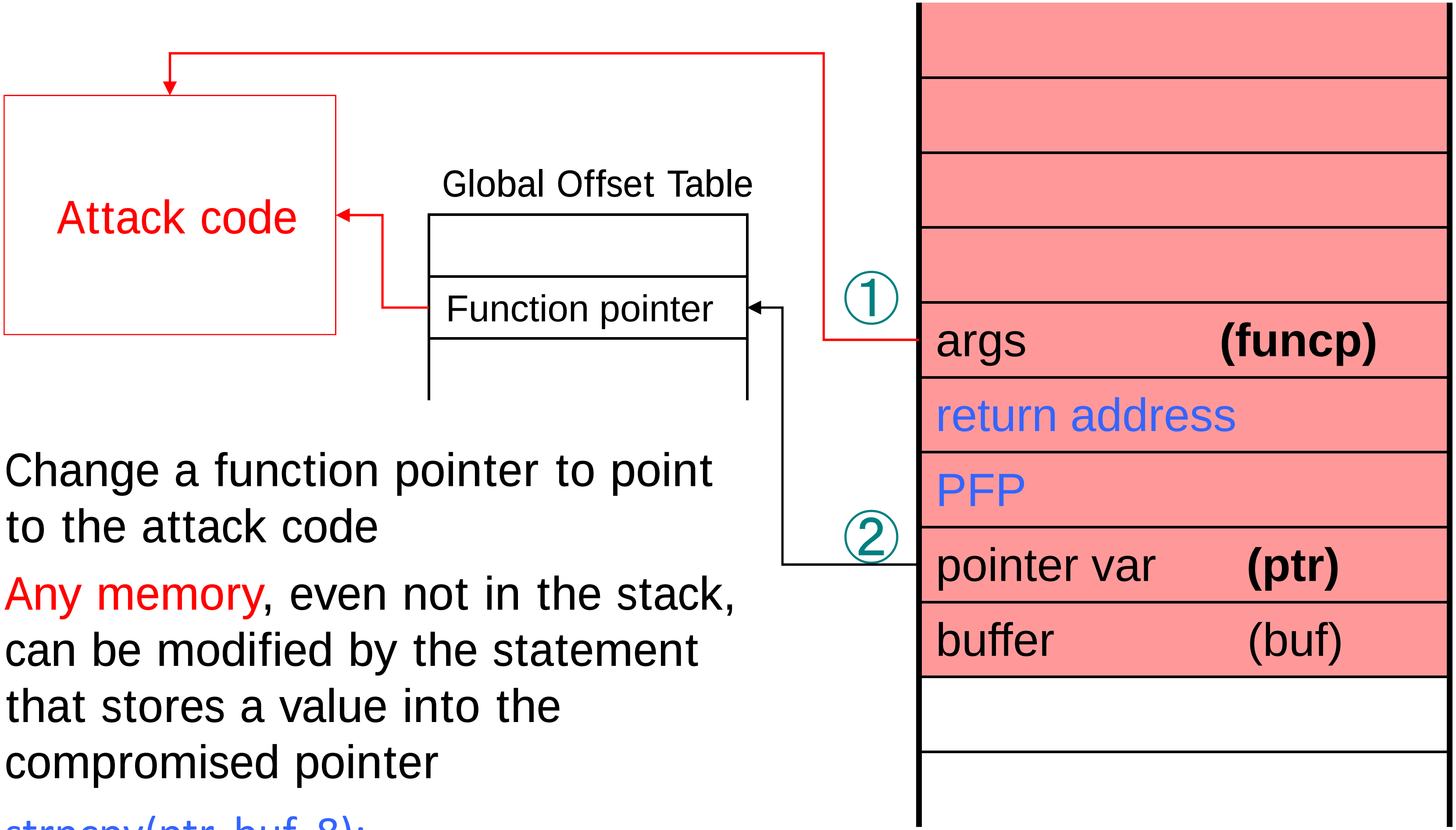
Misuse of strncpy in htpasswd "Fix"

- Published "fix" for Apache htpasswd overflow:

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



Attack #2: Pointer Variables



① Change a function pointer to point to the attack code

② Any memory, even not in the stack, can be modified by the statement that stores a value into the compromised pointer

```
strncpy(ptr, buf, 8);
*ptr = 0;
```

Off-By-One Overflow

- Home-brewed range-checking string copy

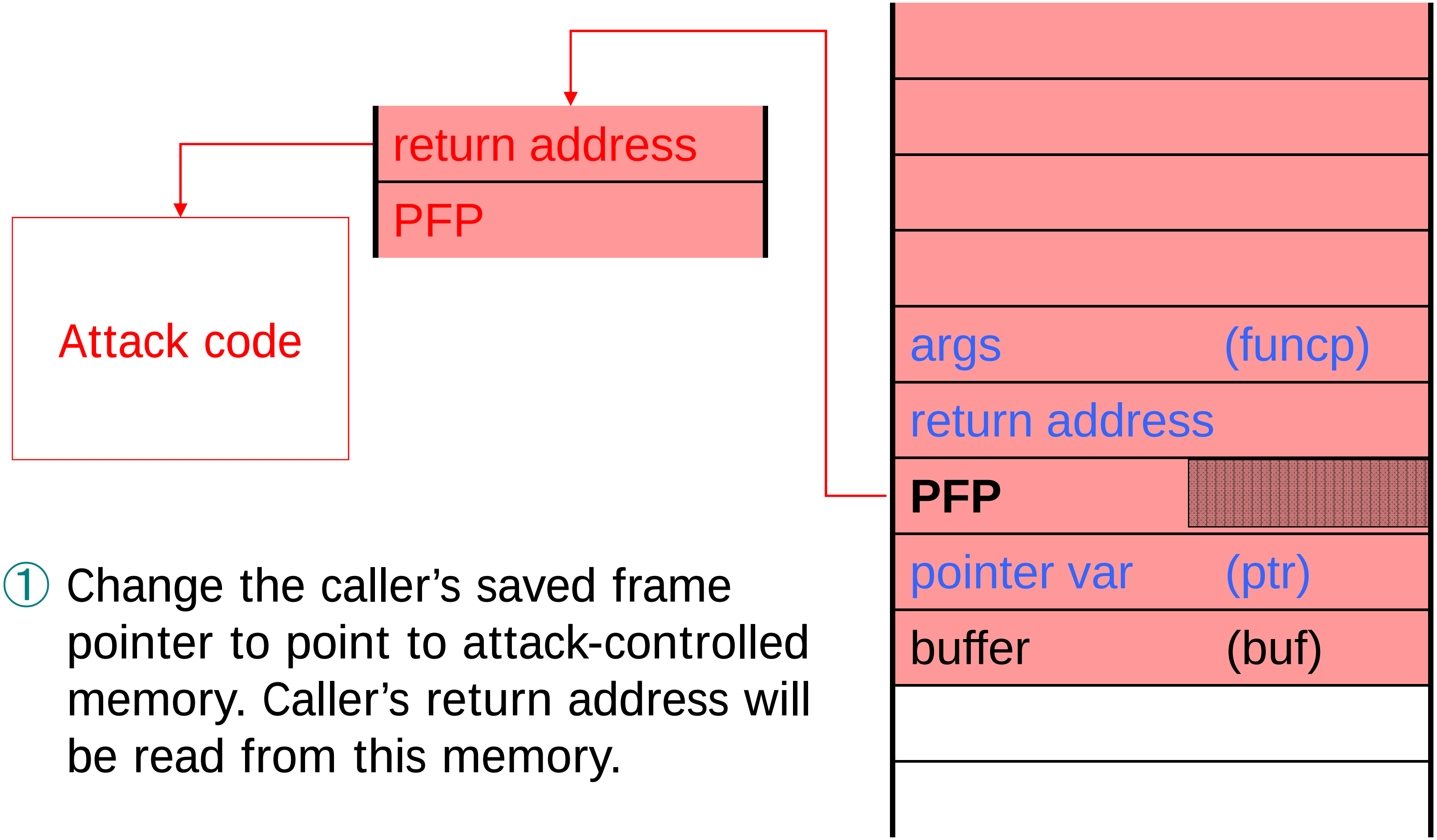
```
void notSoSafeCopy(char *input) {
     char buffer[512]; int i;

     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!

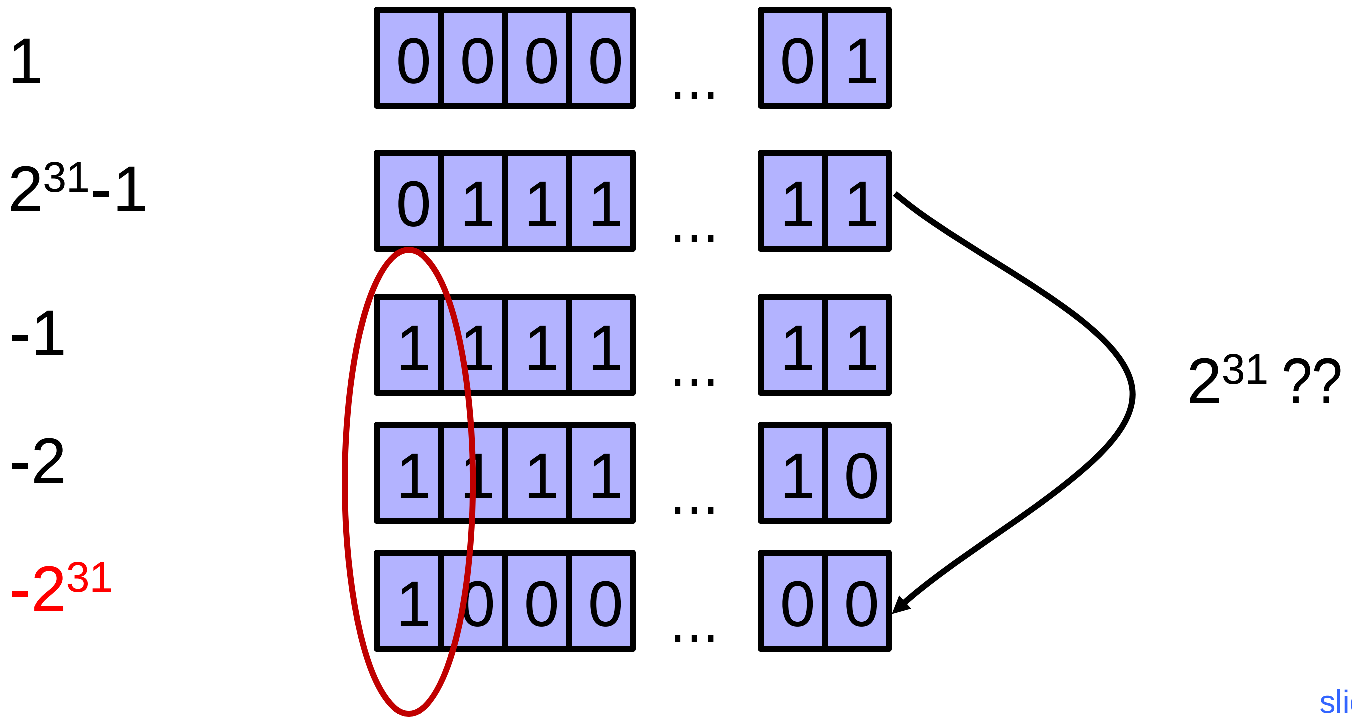
Attack #3: Frame Pointer



① Change the caller's saved frame pointer to point to attack-controlled memory. Caller's return address will be read from this memory.

Two's Complement

- Binary representation of negative integers
- Represent X (where X<0) as $2^N - |X|$
  - N is word size (e.g., 32 bits on x86 architecture)



Integer Overflow

```
static int getpeername1(p, uap, compat) {
// In FreeBSD kernel, retrieves address of peer to which a socket is connected
    ...
    struct sockaddr *sa;
    ...
    len = MIN(len, sa->sa_len);
    ... copyout(sa, (caddr_t)uap->asa, (u_int)len);
    ...
}
```

Checks that “len” is not too big

Negative “len” will always pass this check...

... interpreted as a huge unsigned integer here

Copies “len” bytes from kernel memory to user space

... will copy up to 4G of kernel memory

Heap Overflow

- Overflowing buffers on heap can change pointers that point to important data
  - Sometimes can also transfer execution to attack code
    - For example, December 2008 attack on XML parser in Internet Explorer 7 - see http://isc.sans.org/diary.html?storyid=5458
- Illegitimate privilege elevation: if program with overflow has sysadm/root rights, attacker can use it to write into a normally inaccessible file
  - For example, replace a filename pointer with a pointer into buffer location containing name of a system file
    - Instead of temporary file, write into AUTOEXEC.BAT

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 %-encoded characters

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

Implementation of Variable Args

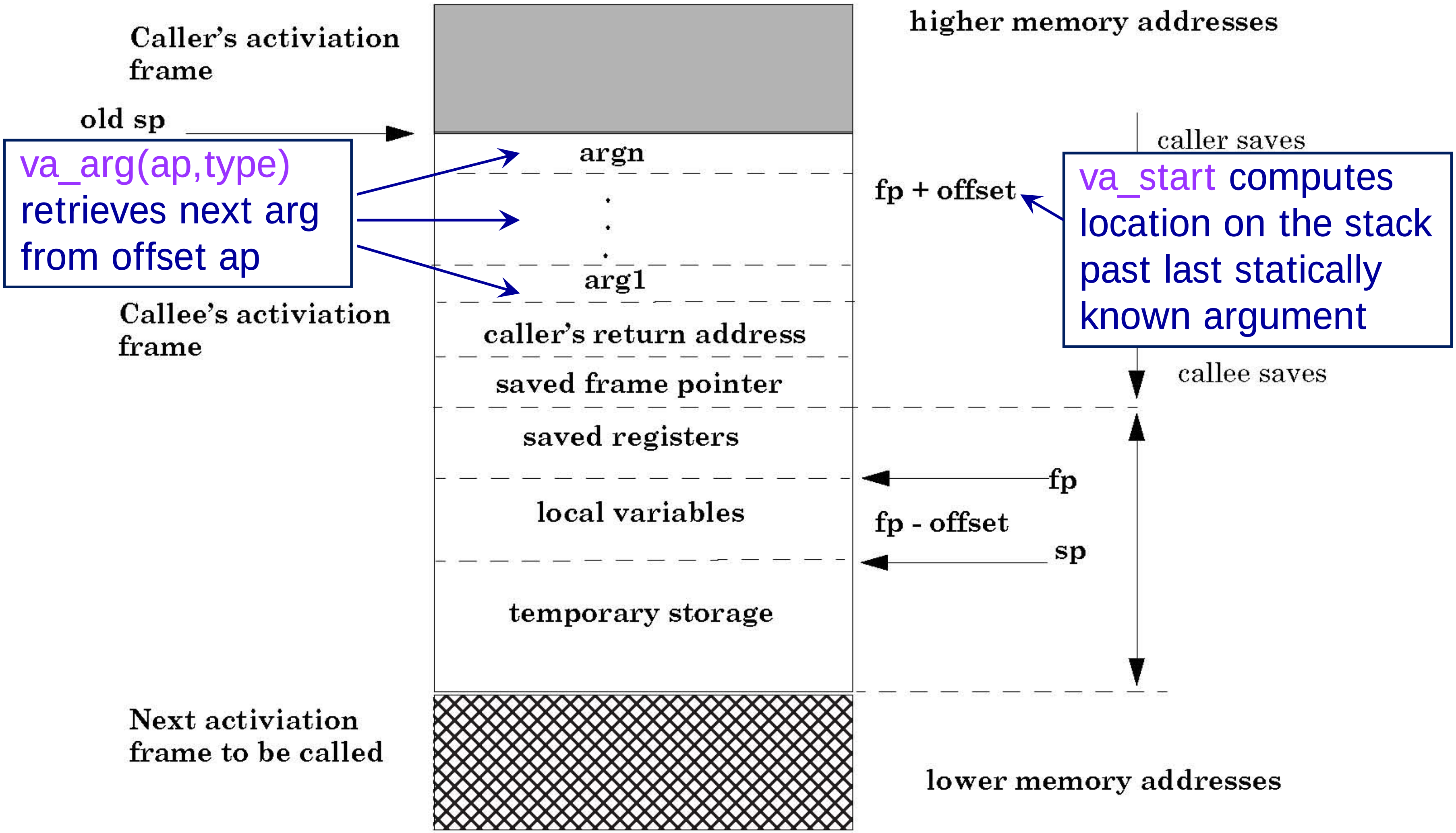
- Special functions va_start, va_arg, va_end compute arguments at run-time (how?)

```
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 */
}
```

Activation Record for Variable Args



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

  – If the buffer contains a format symbol 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!

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); …
```

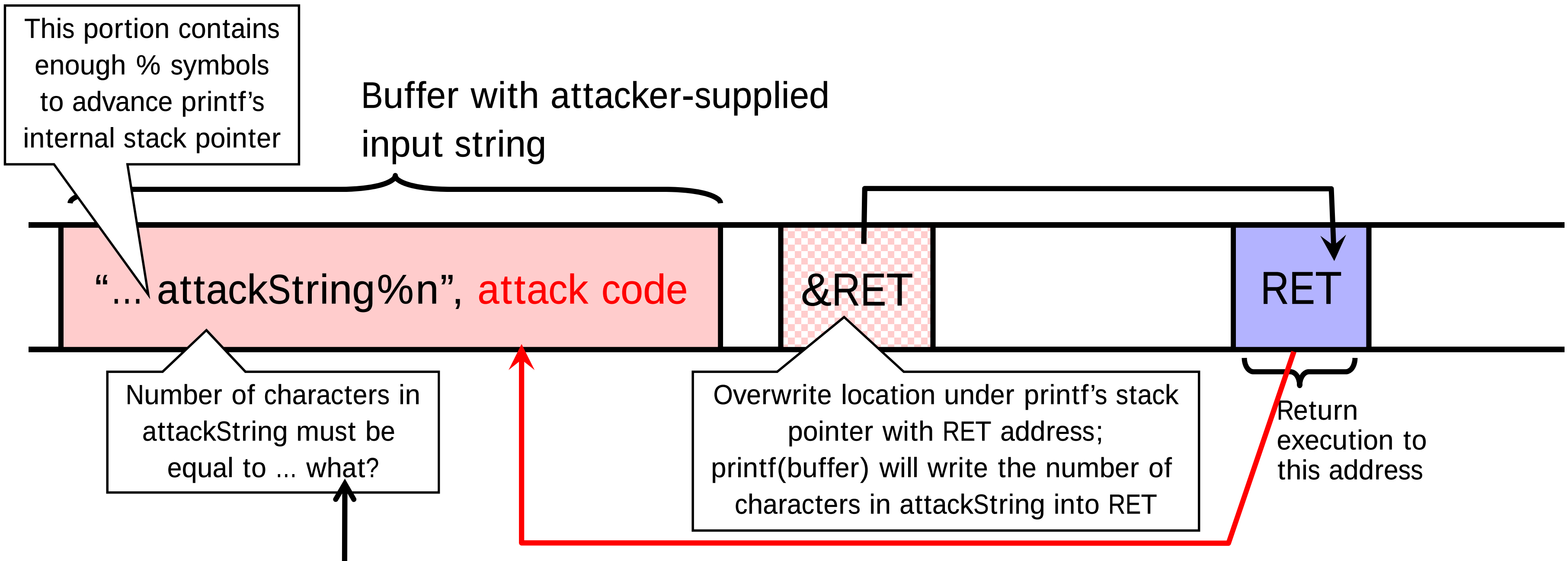
  - Argument of printf is interpeted as destination address
  - This writes 14 into myVar ("Overflow this!" has 14 characters)

- 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 Mung Return Address



C has a concise way of printing multiple symbols: %Mx will print exactly M bytes (taking them from the stack). If attackString contains enough "%Mx" so that its total length is equal to the most significant byte of the address of the attack code, this byte will be written into &RET.

Repeat three times (four "%n" in total) to write into &RET+1, &RET+2, &RET+3, replacing RET with the address of attack code.

- See "Exploting Format String Vulnerabilities" for details

Other Targets of Memory Exploits

- Configuration parameters
  - E.g., directory names that confine remotely invoked programs to a portion of the server's file system
- Pointers to names of system programs
  - For example, replace the name of a harmless script with an interactive shell
  - This is not the same as return-to-libc (why?)
- Branch conditions in input validation code

SSH Authentication Code

```
void do_authentication(char *user, ...) {
1:   int authenticated = 0;
      ...
2:   while (!authenticated) {
         /* Get a packet from the client */
3:       type = packet_read();
         // calls detect_attack() internally
4:       switch (type) {
          ...
5:       case SSH_CMSG_AUTH_PASSWORD:
6:        if (auth_password(user, password))
7:             authenticated =1;
         case ...
         }
8:       if (authenticated) break;
     }
     /* Perform session preparation. */
9:   do_authenticated(pw);
}
```

write 1 here

Loop until one of the authentication methods succeeds

detect_attack() prevents checksum attack on SSH1...

...and also contains an overflow bug which permits the attacker to put any value into any memory location

Break out of authentication loop without authenticating properly