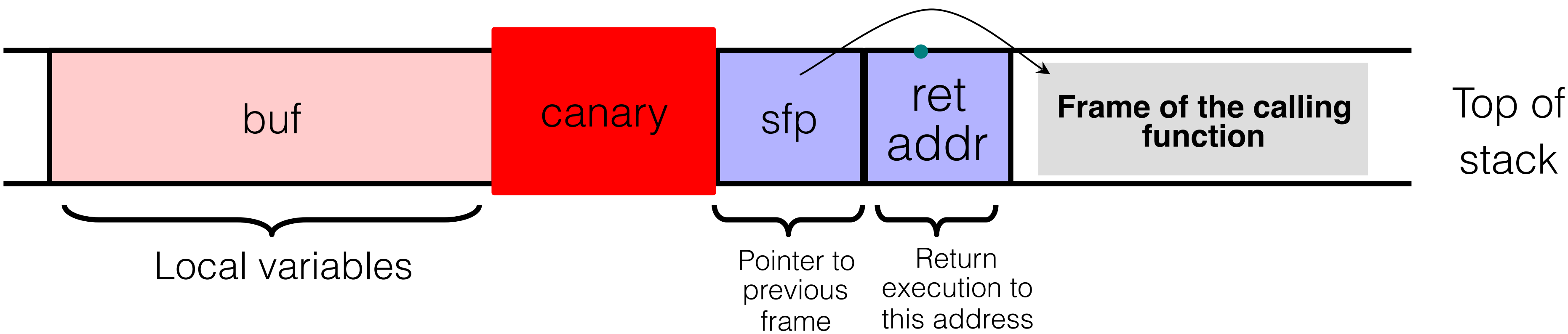
Integer Overflow, Format Strings

Yan Huang

Credits: Vitaly Shmatikov

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



- Candidate Canaries
  - Choose random canary value picked on program start
  - Terminators: "\0", newline, linefeed, EOF

What Can Still Be Overwritten?

- Other string buffers in the vulnerable function
- Any data stored on the stack
  - Exception handling records
  - Pointers to virtual method tables

    C++: call to a member function passes as an argument “this” pointer to an object on the stack

    Stack overflow can overwrite this object’s vtable pointer and make it point into an attacker-controlled area

    When a virtual function is called (how?), control is transferred to attack code (why?)

    Do canaries help in this case?

    (Hint: when is the integrity of the canary checked?)

Example of Failed StackGuard — Litchfield's Attack

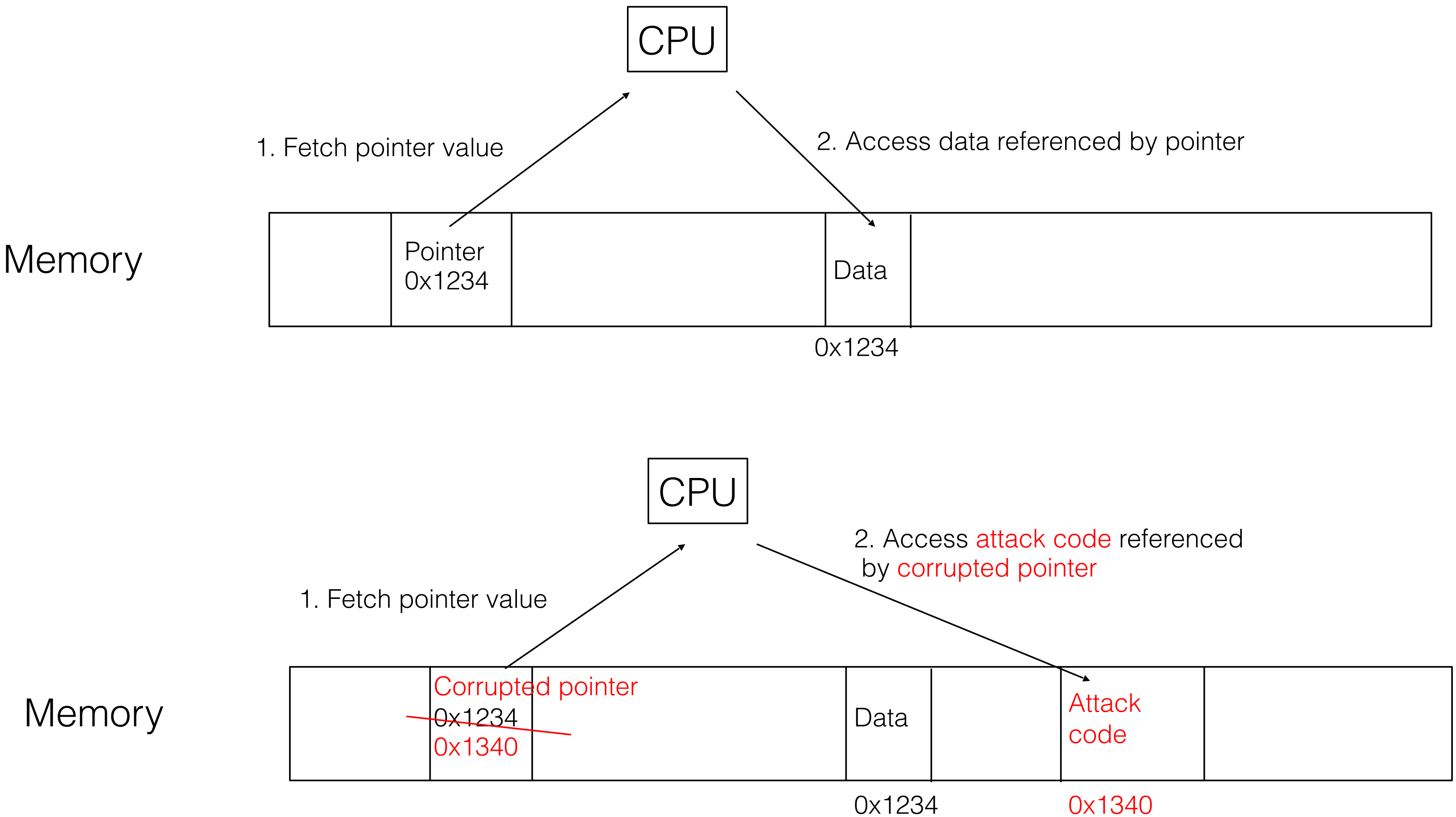
- Microsoft Windows 2003 server implements several defenses against stack overflow
  - Random canary (with /GS option in the .NET compiler)
  - When canary is damaged, exception handler is called
  - Address of exception handler stored on stack above RET
- Attack: smash the canary AND overwrite the pointer to the exception handler with the address of the attack code
  - Attack code must be on heap and outside the module, or else Windows won't execute the fake "handler"
  - Similar exploit used by CodeRed worm

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 overflown 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

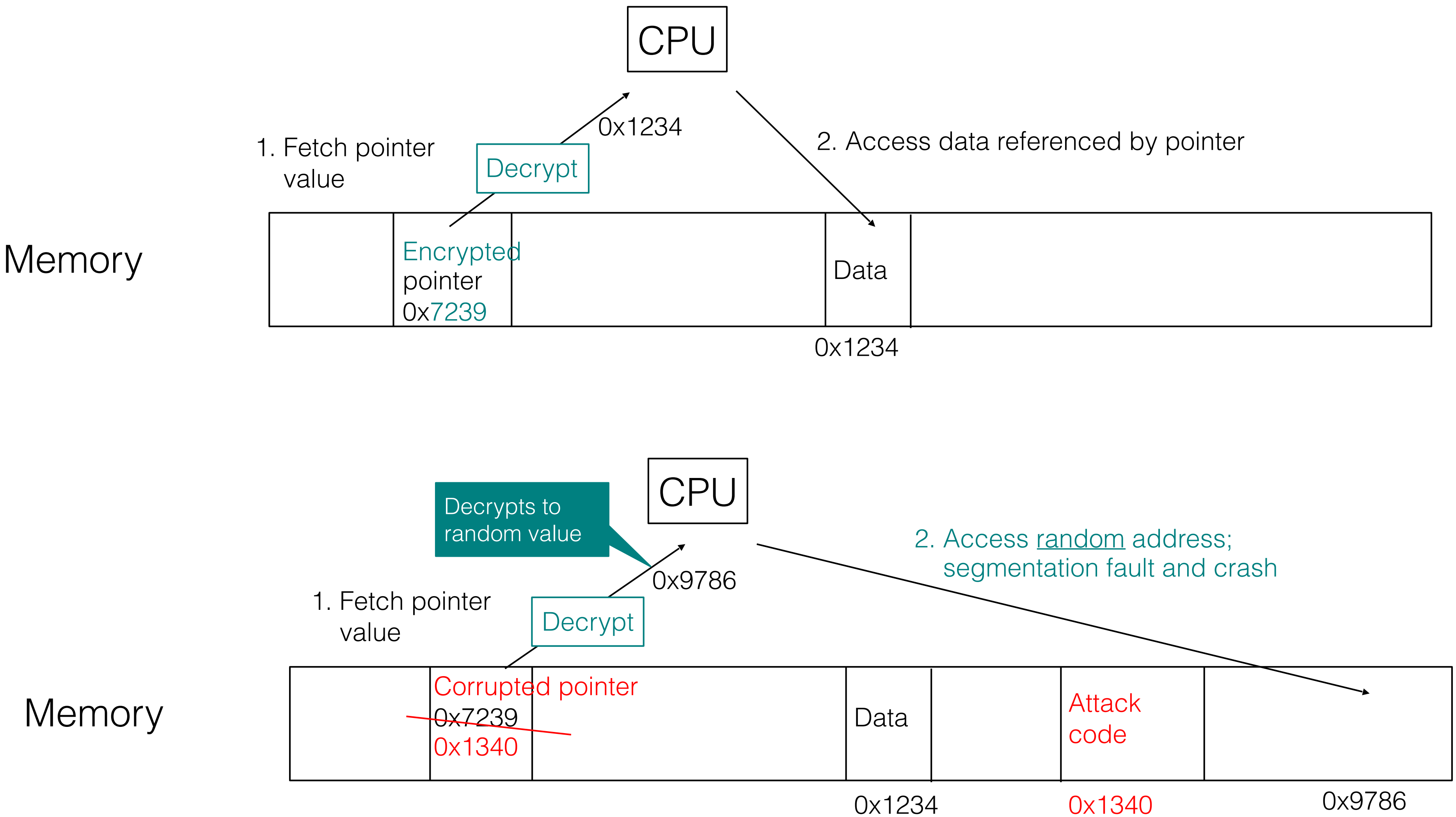
Normal Pointer Dereference

[Cowan]



PointGuard Dereference

[Cowan]

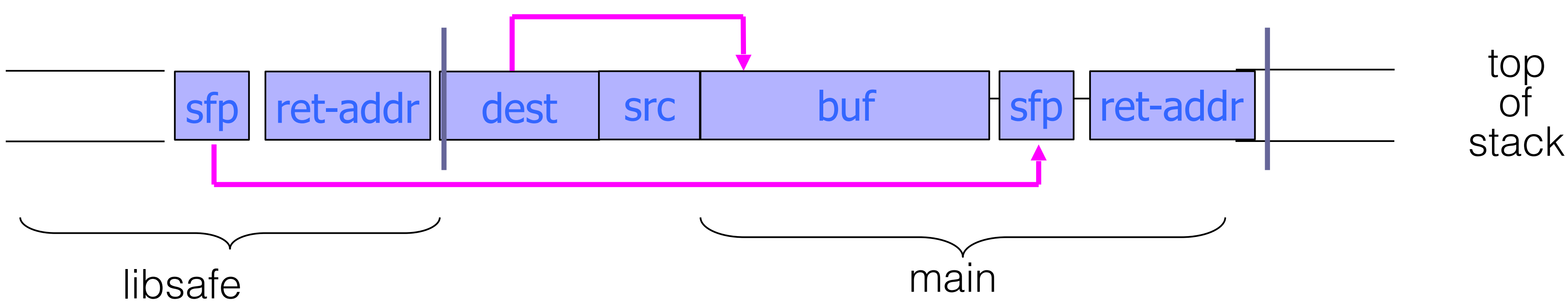


PointGuard Issues

- 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 the key in a memory page inaccessible to adversaries
- PG’d code doesn’t mix well with normal code
  - What if PG’d code needs to pass a pointer to OS kernel?

Libsafe

- Intercepts calls to strcpy(dest, src) and other unsafe C library functions
  - Checks if there is sufficient space in current stack frame |framePointer – dest| > strlen(src)
  - If yes, does strcpy; else terminates application
- Dynamically loaded library – no need to recompile!



Limitations of Libsafe

- Protects frame pointer and return address from being overwritten by a stack overflow
- Does not prevent sensitive local variables below the buffer from being overwritten
- Does not prevent overflows on global and dynamically allocated buffers

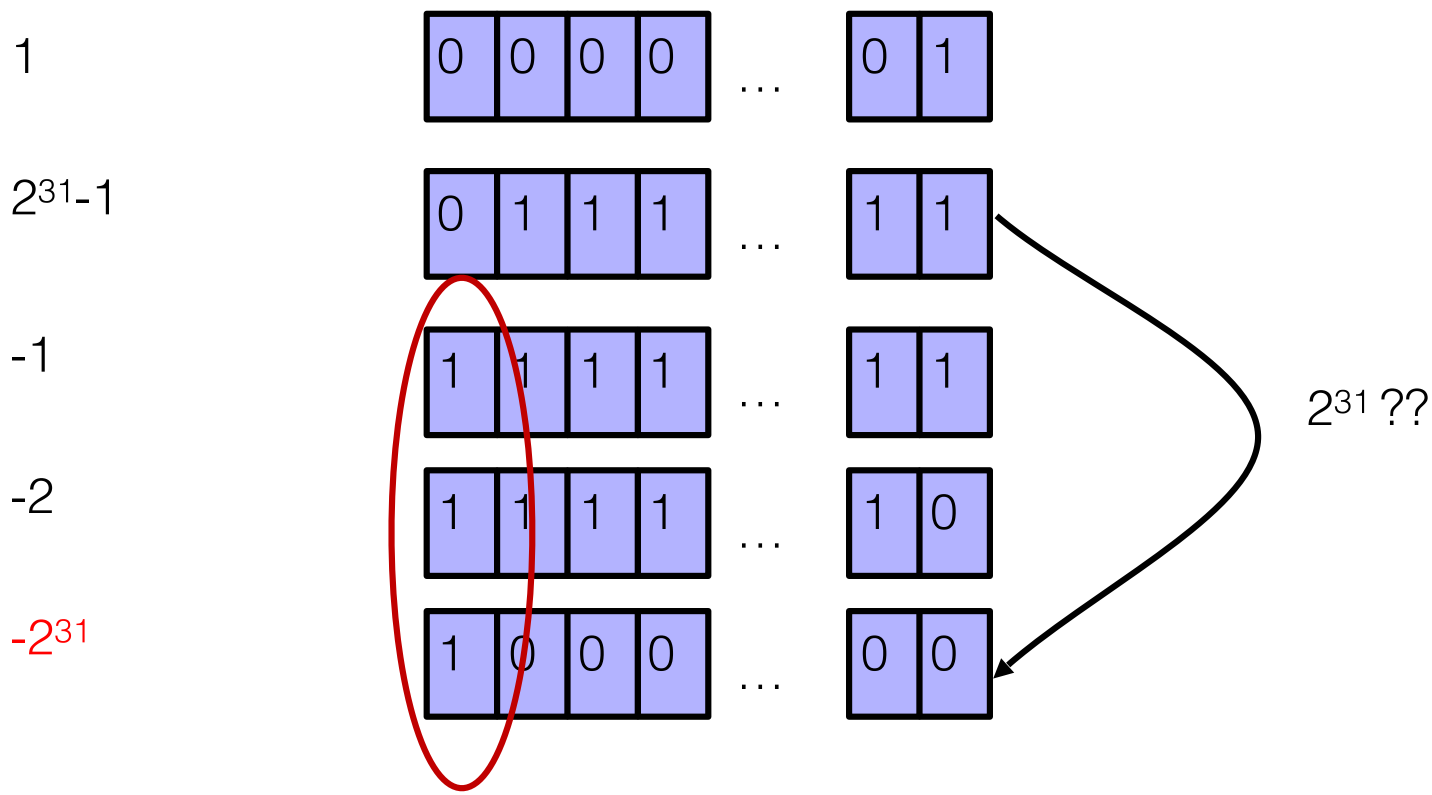
Integer Overflow Attacks

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;
     …
    assert(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

… will copy up to 4G of kernel memory

Copies “len” bytes from kernel memory to user space

Format String Attacks

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

Implementation of Variable Args

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

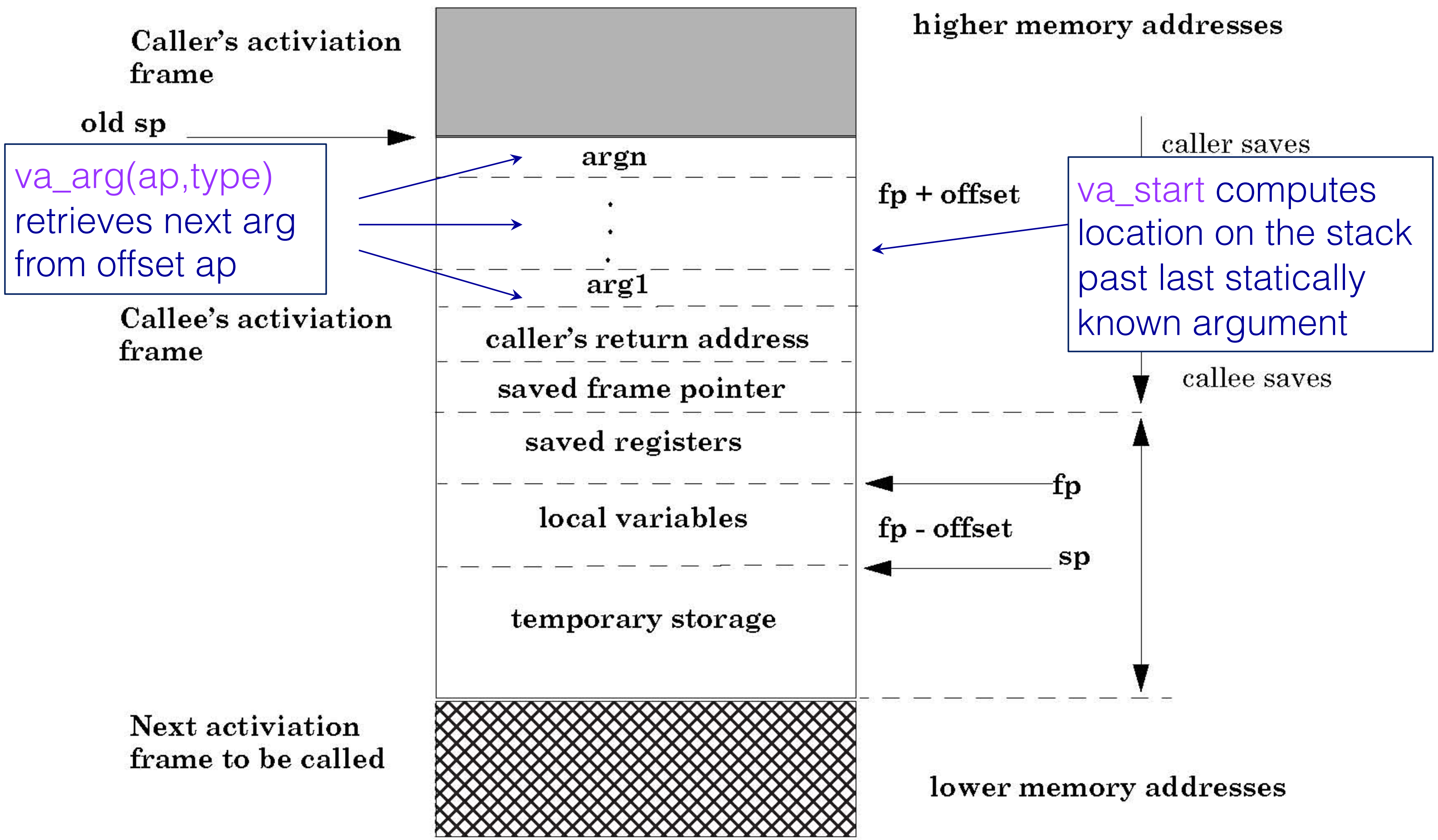
```
void printf(const char* format, ...)
{
      int i; char c; char* s; double d;
      va_list ap;  /* declare an "argument pointer" to a variable arg list */
      va_start(ap, format);  /* initialize arg pointer using last known arg */

      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

Frame with 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! (how?)

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 interpreted as destination address

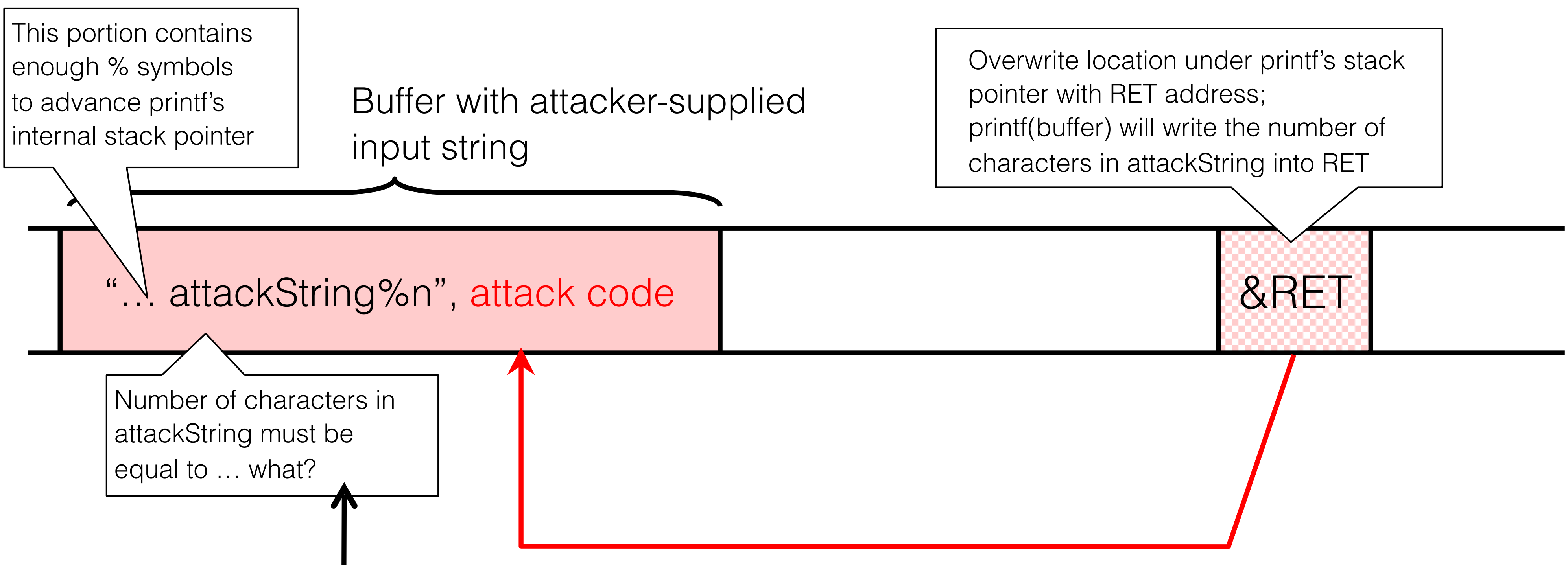
  This writes 14 into myVar ("Overflow this!" has 14 characters)

- 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 4M bytes (taking them from the stack). Attack string should contain enough "%Mx" so that the number of characters printed is equal to the most significant byte of the address of the attack code.

See "Exploiting Format String Vulnerabilities" for details