Security of memory allocators for C and C++

Yves Younan
Wouter Joosen
Frank Piessens
Hans Van den Eynden

Report CW 419, July 2005

Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

Security of memory allocators for C and C++

Yves Younan
Wouter Joosen
Frank Piessens
Hans Van den Eynden

Report CW 419, July 2005

Department of Computer Science, K.U.Leuven

Abstract

Most memory allocators ignore security issues. Instead they focus on maximizing performance and limiting fragmentation and waste. While these are very important issues for a memory allocator, in the days of worms that use code injection attacks to cause significant economical damage, security can not be ignored. This paper evaluates a representative set of commonly used memory allocators for C and C++ with respect to their resilience against code injection attacks. We present a significant improvement for memory allocators in order to increase robustness against code injection attacks. We evaluate this new approach in terms of performance and memory usage and show that the associated overhead is negligible.

CR Subject Classification : K6.5, A.1, D3.4, D4.2

1 Introduction

Security has become an important concern for all computer users. Worms and hackers are a part of every day internet life. A particularly dangerous attack is the code injection attack, where attackers are able to insert code into the program's address space and can subsequently execute it. Programs written in C are particularly vulnerable to such attacks. Attackers can use a range of vulnerabilities to inject code. The most well known and most exploited is of course the standard buffer overflow: attackers write past the boundaries of a stack-based buffer and overwrite the return address of a function and point it to their injected code. When the function subsequently returns, the code injected by the attackers is executed [1].

These are not the only kind of code injection attacks though: a buffer overflow can also exist on the heap, allowing an attacker to overwrite heap-stored data. As pointers are not always available in normal heap-allocated memory, attackers often overwrite the management information that the memory allocator relies upon (to function correctly). A double free vulnerability, where a particular part of heap-allocated memory is deallocated twice could also be used by an attacker to inject code.

Many countermeasures have been devised that try to prevent these kind of attacks [24]. However many have focussed on preventing stack-based buffer overflows and only few have concentrated on protecting the heap or memory allocators from attack.

In this paper we evaluate some commonly used memory allocators for C and C++ with respect to their resilience against code injection attacks and present a significant improvement for memory allocators in order to increase robustness against code injection attacks. Our prototype implementation comes at a very modest cost in both performance and memory usage overhead.

The paper is structured as follows: section 2 explains which vulnerabilities can exist for heap-allocated memory. Section 3 describes several memory allocators and how they can be exploited by an attacker using one of the vulnerabilities of section 2 to perform code injection attacks. Section 4 describes our new more robust approach to handling the management information associated with chunks of memory. In section 6 related work in improving security for memory allocators is discussed. A synthesis of which allocators are vulnerable to which attacks is presented in section 7. Section 5 contains the results of tests in which we compare our memory allocator to the original allocator in terms of performance overhead and memory usage. Finally, section 8 describes our future work and presents our conclusion.

2 Heap-based vulnerabilities for code injection attacks

There are a number of vulnerabilities that occur frequently and as such have become a favorite for attackers to use to perform code injection. We will examine how different memory allocators might be misused by using one of three common vulnerabilities: "heap-based buffer overflows", "off by one errors" and "dangling pointer references". In this section we will describe what these vulnerabilities are and how they could lead to a code injection attack.

2.1 Heap-based buffer overflow

Heap memory is dynamically allocated at run-time by the application. The buffer overflow, which is usually exploited on the stack, is also possible in this kind of memory. However exploitation of such heap-based buffer overflows usually relies on finding either function pointers or by performing an indirect pointer attack [7] on data pointers in this memory area, but these pointers are not always present. As such most attackers overwrite the memory management information that the memory allocator stores in or around memory chunks it manages. By modifying this information, attackers can perform an indirect pointer overwrite. This allows attackers to overwrite arbitrary memory locations, which could lead to a code injection attack [2, 23]. In the following sections we will describe how an attacker could use specific memory allocators to perform this kind of attack.

2.2 Off by one/five errors

An off by one error is a special case of the buffer overflow. When an off by one occurs, the adjacent memory location is overwritten by exactly one byte. This often happens when a programmer loops through an array but typically ends at the array's size rather than stopping at the preceding element (because arrays start at 0). In some cases these errors can also be exploitable by an attacker [2, 23]. A more generally exploitable version of the off by one for memory allocators is an off by five, while these do not occur that often in the wild, they demonstrate that it is possible to cause a code injection attack when little memory is available. These errors are usually only exploitable on little endian machines because the least significant byte of an integer is stored before the most significant byte in memory.

2.3 Dangling pointer references

Dangling pointers are pointers to memory locations that are no longer allocated. In most cases this will lead to a program crash. However when it happens in heap memory, it could lead to a double free vulnerability, where a memory location is freed twice. Such a double free vulnerability could be misused by an attacker to modify the management information associated with a chunk and as a result could lead to a code injection attack [9, 23]. This kind of vulnerability is not present in all memory allocators, as some will check if a chunk is free or not before freeing it a second time. In the following sections we will describe if and how specific memory allocators could be exploited using this vulnerability.

More information about these attacks can be found in [23].

3 Memory allocators

In this section we will examine a representative set of allocators. We have chosen these specific allocators because they are in common use and easily available for review. Boehm's garbage collector was chosen to determine whether a representative memory manager for C/C++ would be more resilient against attack.

We will describe how these allocators work in normal circumstances and then will explain how a heap-vulnerability which can overwrite the management information of these memory allocators could be used by an attacker to cause a code injection attack. We will use the same structure to describe all memory allocators: first we describe how the allocator works and afterwards we examine if and how an attacker could exploit it to perform code injection attacks (given one of the aforementioned vulnerabilities exists).

3.1 Doug Lea's memory allocator

Doug Lea's memory allocator [16, 17] (commonly referred to as dlmalloc) was designed as a general purpose memory allocator which could be used by any kind of program. It is used as a basis for the allocator in the GNU/Linux operating system. The version of the allocator described in this section is based on version 2.7.2.

3.1.1 Description

The memory allocator divides the heap memory at its disposal into contiguous chunks[1], which vary in size as the various allocation routines (*malloc*, *free*, *realloc*, ...) are called. An invariant is that a free chunk never borders another free chunk when one of these routines has completed: if two free chunks had bordered, they would have been coalesced into one larger free chunk. These free chunks are kept in sorted doubly linked lists of the same size (for small chunks) or of an interval of size (for larger chunks), which are accessed via an array (whose elements are called bins). When the memory allocator at a later time requests a chunk of the same size as one of these free chunks, the first chunk of appropriate size will be removed from the list and will be made available for use in the program (i.e. it will turn into an allocated chunk).

Chunk structure Memory management information associated with a chunk is stored in-band. Figure 1 illustrates what a heap of used and unused chunks could look like. *Chunk1* is an allocated chunk containing information about the size of the chunk stored before it and its own size[2]. The rest of the chunk is available for the program to write data in. *Chunk2*[3] is a free chunk: it is stored

[1] A chunk is a block of memory that is allocated by the allocator, it can be larger than what programmer requested because it usually reserves space for management information

[2] The size of allocated chunks is always a multiple of eight, so the three least significant bits of the size field are used for management information: a bit to indicate if the previous chunk is in use or not and one to indicate if the memory is mapped or not. The last bit is currently unused. The "previous chunk in use"-bit can be modified by an attacker to force coalescing of chunks. How this coalescing can be abused is explained later.

[3] The representation of *chunk2* is not entirely correct: if *chunk1* is in use, it will be used to store 'user data' for *chunk1* and not the size of *chunk1*. We have chosen to represent *chunk2* this way as this detail is not relevant to the discussion here, see section 3.1.2.

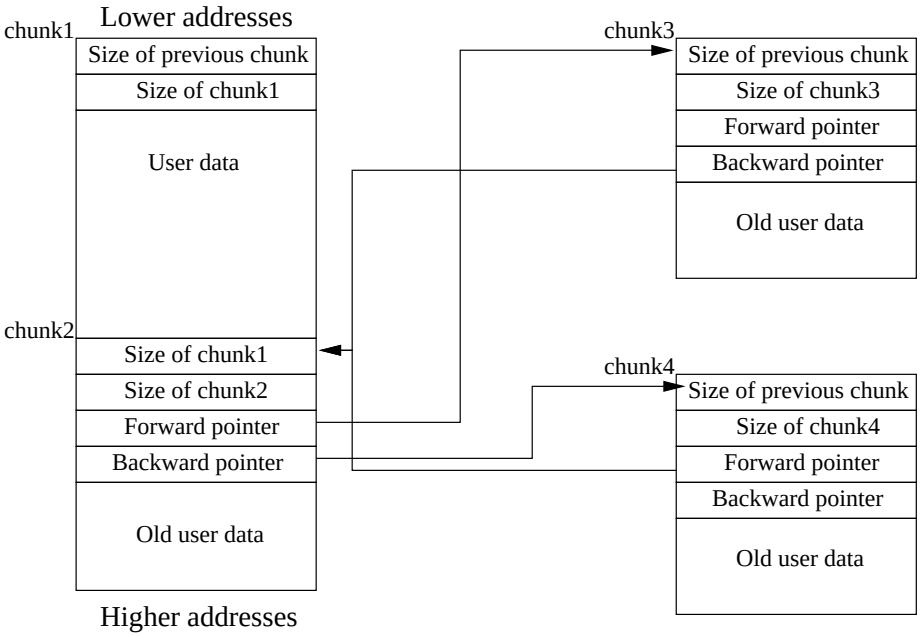

Figure 1: Heap containing used and free chunks

in the doubly linked list of chunks of equal size. The information for this list is stored over the first 8 bytes of what was user data when the chunk was allocated. This particular chunk is located between *chunk3* in *chunk4* in the list. *Chunk3* is the first chunk in the chain: its backward pointer points to *chunk2* and its forward pointer points to a previous chunk in the list. *Chunk2* is the next chunk, with its forward pointer pointing to *chunk3* and its backward pointer pointing to *chunk4*. *Chunk4* is the last chunk in our example: its backward pointer points to a next chunk in the list and its forward pointer points to *chunk2*.

3.1.2 Exploitation

Dlmalloc is vulnerable to all three of the previously described vulnerabilities [2, 9, 11, 21]. Here we will describe how these vulnerabilities may lead to a code injection attack.

Overwriting memory management information Figure 2 shows what could happen if an array that is located in *chunk1* is overflowed: an attacker has overwritten the management information of *chunk2*. The size fields are left unchanged (although these could be modified if needed). The forward pointer has been changed to point to 12 bytes before the return address and the backward pointer has been changed to point to code that will jump over the next few bytes. When *chunk1* is subsequently freed, it will be coalesced together with chunk2 into a larger chunk. As *chunk2* will no longer be a separate chunk after the coalescing it must first be removed from the list of free chunks. The *unlink* macro takes care of this: internally a free chunk is represented by a struct containing the following unsigned long integer fields (in this order): *prev_size*,

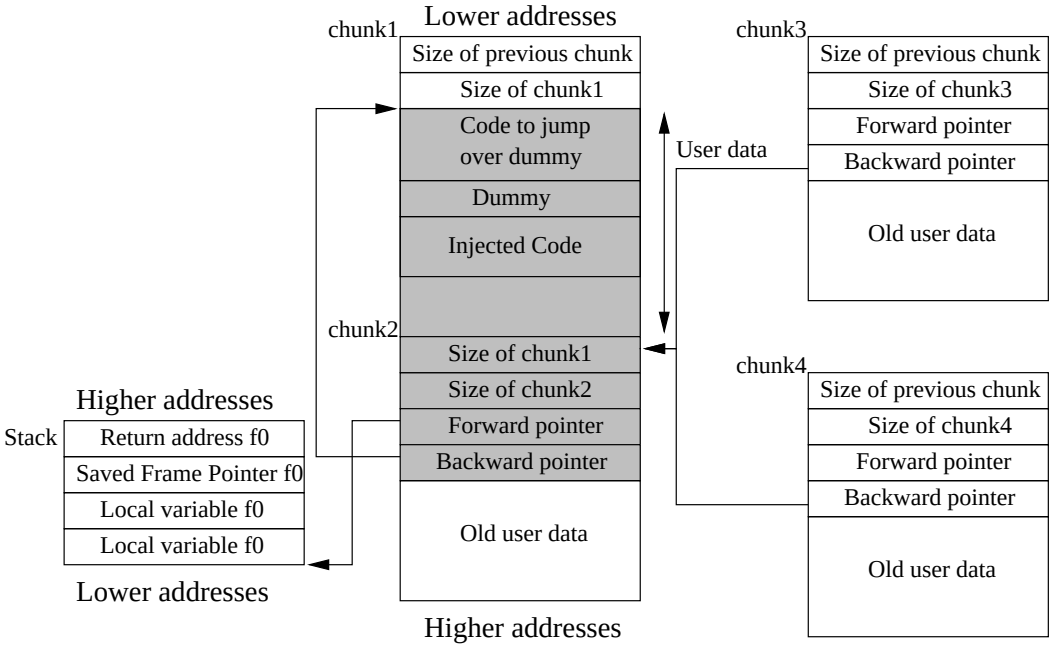

Figure 2: Heap-based buffer overflow in dlmalloc

size, *fd* and *bk*. A chunk is unlinked as follows:

```
chunk2->fd->bk = chunk2->bk
chunk2->bk->fd = chunk2->fd
```

Which is the same as (based on the struct used to represent malloc chunks):

```
*(chunk2->fd+12) = chunk2->bk
*(chunk2->bk+8) =  chunk2->fd
```

As a result, the value of the memory location that is twelve bytes after the location that *fd* points to will be overwritten with the value of *bk*, and the value of the memory location eight bytes after the location that *bk* points to will be overwritten with the value of *fd*. So in the example in Figure 2 the return address would be overwritten with a pointer to code that will jump over the place where *fd* will be stored and will execute code that the attacker has injected. However, since the eight bytes after the memory that bk points to will be overwritten with a pointer to fd (illustrated as dummy in Figure 2), the attacker needs to insert code to jump over the first twelve bytes into the first eight bytes of his injected code. Using this technique an attacker could overwrite arbitrary memory locations [2, 11, 21].

Off by one/five error An off by one error could also be exploited in the Doug Lea's memory allocator [2]. If the chunk is located immediately next to the next chunk (i.e. not padded to be a multiple of eight), then an off by one can be exploited: if the chunk is in use, the prev_size field of the next chunk will be used for data and by writing a single byte out of the bounds

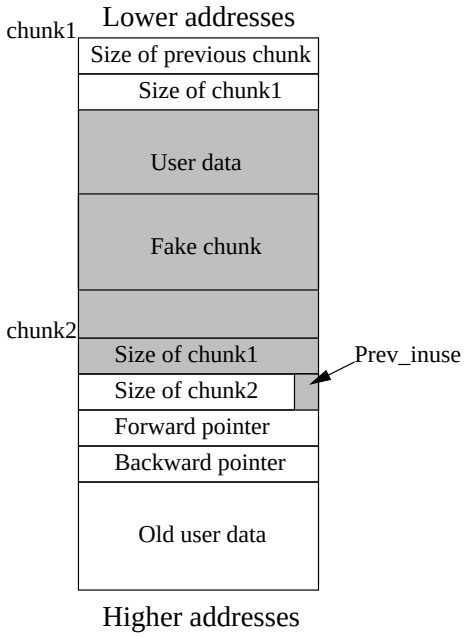

Figure 3: Off by one error: size of chunk1 is part of data of chunk1 and the least significant bit of size of chunk2 has been overwritten to unset the prev_inuse flag

of the chunk, the least significant byte of the size field of the next chunk will be overwirtten. As the least significant byte contains the prev_inuse bit, the attacker can make the allocator think the chunk is free and will coalesce it when the second chunk is freed. Figure 3 depicts the exploit: the attacker creates a fake chunk in the *chunk1* and sets the prev_size field accordingly and overwrites the least significant byte of *chunk2*'s size field to mark the current chunk as free. The same technique using the forward and backward pointers that was used in section 3.1.2 can now be used to overwrite arbitrary memory locations.

Double free Dlmalloc can be used for a code injection attack if a double free exists in the program [9]. Figure 4 illustrates what happens when a double free occurs. The full lines in this figure are an example of what the list of free chunks of memory might look like when using this allocator.

Chunk1 is bigger than the *chunk2* and *chunk3* (which are both the same size), meaning that *chunk2* is the first chunk in the list of free chunks of equal size. When a new chunk of the same size as *chunk2* is freed, it is placed at the beginning of this list of chunks of the same size by modifying the backward pointer of *chunk1* and the forward pointer of *chunk2*.

When a chunk is freed twice it will overwrite the forward and backward pointers and could allow an attacker to overwrite arbitrary memory locations at some later point in the program. As mentioned in the previous section: if a new chunk of the same size as *chunk2* is freed it will be placed before *chunk2* in the list. The following pseudo code demonstrates this (modified from the original version found in dlmalloc):

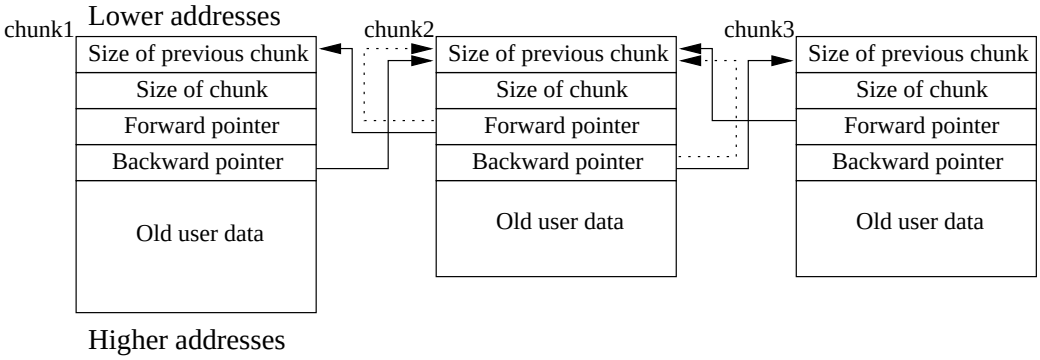

Figure 4: List of free chunks: full lines show a normal list of chunks, dotted lines show the changes after a double free has occurred.

```
BK = front_of_list_of_same_size_chunks
FD = BK->FD
new_chunk->bk = BK
new_chunk->fd = FD
FD->bk = BK->fd = new_chunk
```

The backward pointer of *new_chunk* is set to point to *chunk2*, the forward pointer of this backward pointer (i.e. *chunk2->fd = chunk1*) will be set as the forward pointer for *new_chunk*. The backward pointer of the forward pointer (i.e. *chunk1->bk*) will be set to *new_chunk* and the forward pointer of the backward pointer (*chunk2->fd*) will be set to *new_chunk*.

If chunk2 would be freed twice the following would happen (substitutions made on the code listed above):

```
BK = chunk2
FD = chunk2->fd
chunk2->bk = chunk2
chunk2->fd = chunk2->fd
chunk2->fd->bk = chunk2->fd = chunk2
```

The forward and backward pointers of *chunk2* both point to itself. The dotted lines in Figure 4 illustrate what the list of free chunks looks like after a second free of *chunk2*.

```
chunk2->fd->bk = chunk2->bk
chunk2->bk->fd = chunk2->fd
```

But since both *chunk2->fd* and *chunk2->bk* point to *chunk2*, it will again point to itself and will not really be unlinked. However the allocator assumes it has and the program is now free to use the user data part (everything below 'size of chunk' in Figure 4) of the chunk for its own use.

Attackers can now use the same technique that we previously discussed to exploit the heap-based overflow (see Figure 2): they set the forward pointer to point 12 bytes before the return address and change the value of the backward pointer to point to code that will jump over the bytes that will be overwritten. When the program tries to allocate a chunk of the same size again (or tries to

free this one), it will again try to unlink *chunk2* which will overwrite the return address with the value of *chunk2's* backward pointer.

3.2 CSRI memory allocator

The CSRI memory allocator was designed by Mark Moraes [18]. The description in this section is based on version 1.18 of the allocator. Besides normal operating mode it has extensive debugging modes that will detect memory leaks, heap-based buffer overflows and double frees. We will focus on normal operation.

3.2.1 Description

Chunk structure The chunks used by CSRI store the same information as the chunks in dlmalloc, but it stores them in a different location in the chunk. Every chunk contains a size pointer at the beginning of the chunk and one at the end of the chunk. The least significant bit of the size field is used to denote if the chunk is free or not. The prev and next pointers are also only present in free chunks but are located at the end of the chunk, right before the size field. Figures 5 and 6 show what a used and free chunk look like. When a chunk is freed all the bytes stored in it will be overwritten with a special value: '\125'.

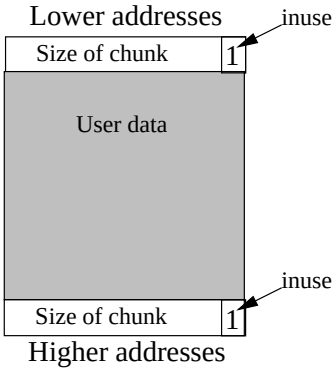

Figure 5: Used CSRI chunk

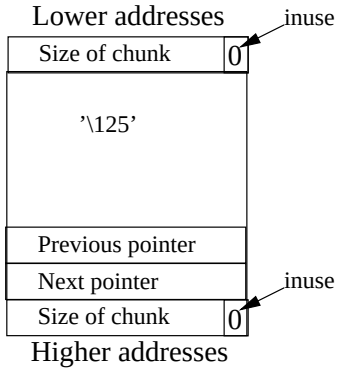

Figure 6: Free CSRI chunk

Free chunks are stored in a circular doubly linked list stored in the *prev* and *next* pointers of the free chunks. To find an empty chunk the allocator uses a next fit algorithm. A small array containing pointers (called rovers) to free chunks smaller than a particular size is used to search for chunks. When a chunk is freed, the allocator will check its size and will place it in the list immediately after the correct rover. And the rover will be made to point to the new chunk. So the starting point of searches in the list is constantly moved and varies depending on the chunk's size.

3.2.2 Exploitation

CSRI will check if a chunk is already free before freeing it, so a simple double free vulnerability (without the possibility of overwriting management information) is not exploitable.

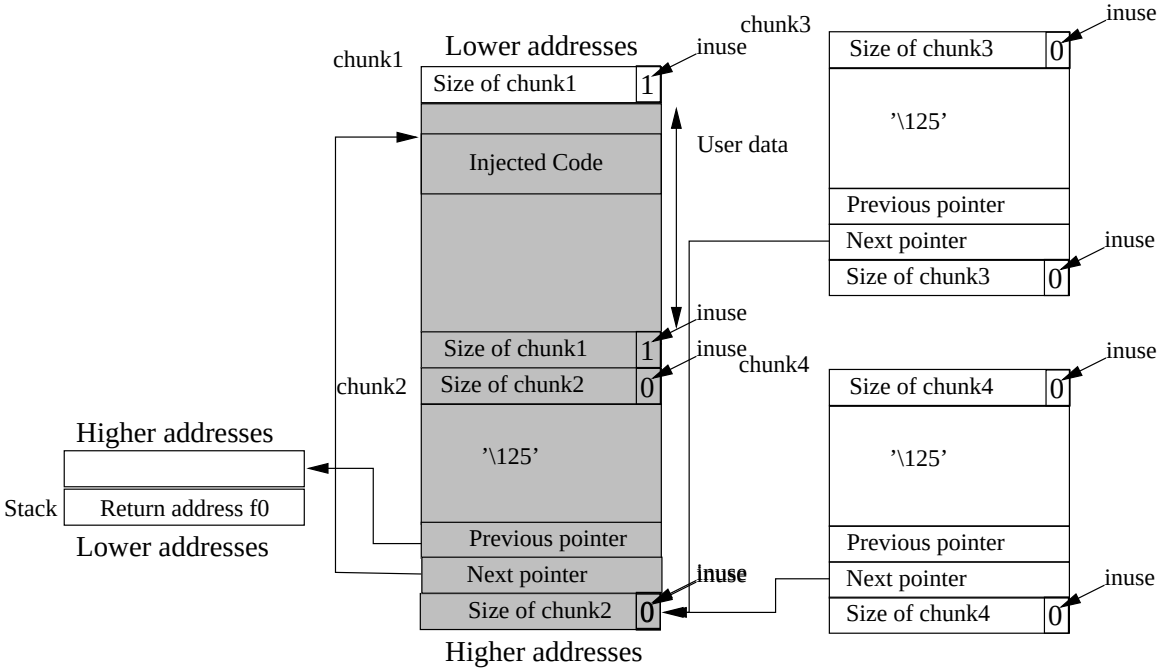

Figure 7: Heap-based buffer overflow in CSRI.

Overwriting memory management information Exploitation of a heap-based buffer overflow against the CSRI memory allocator is closely related to the attack on dlmalloc. The allocators also keeps its free chunks in a doubly linked list of free chunks.

A similar attack technique can be used: by modifying the prev and next pointers. The details are slightly different though: a chunk is removed from the list of free chunks by using the following code:

```
nextp = NEXT(P);
prevp = PREV(P);
NEXT(prevp) = nextp;
PREV(nextp) = prevp;
```

While *NEXT(P)* is defined as *((P) − 1)* and *PREV(P)* is defined as *((P −2))* This results in the following code:

```
*(P−2) − 1 = P − 1
*(P−1) − 2 = P − 2
```

Figure 7 shows how an attacker would exploit this in practice: prev is made to point to the address before the return address on the stack while next is made to point to the injected code. *Prev − 1* (the return address) will be replaced with the value in next while *next − 2* will be replaced with the value in prev. This simplifies the attack as oppposed to dlmalloc because the attacker must no longer insert code to jump over the value that the dlmalloc unlinker would overwrite.

Because the attacker must overwrite the pointers at the end of the following chunk and because the unlinker uses the size info at the end of a chunk rather than at the beginning, neither an off by one nor an off by five attack is feasible.

3.3 Quickfit memory allocator

Quickfit malloc is a memory allocator implemented by Dirk Grunwald [10]. It is based on an old version dlmalloc and uses an implmentation of the quickfit algorithm by [22] for managing memory.

3.3.1 Description

The quickfit memory allocator shares many features with dlmalloc. As such we will only shortly describe the similarities and will describe the differences in more detail.

Chunk structure The chunk structure is a hybrid between the chunk structure of CSRI and the structure of dlmalloc (see Figure 10). It also contains size fields both at the beginning and the end of the chunk and the least significant bit of the size fields is used to denote if the chunk is free or not. The forward and backward pointers are stored at the beginning of free chunks, right after the size field.

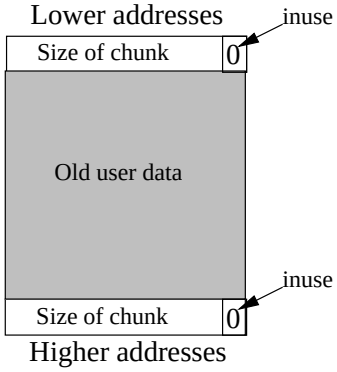

Figure 8: Used Quickfit chunk

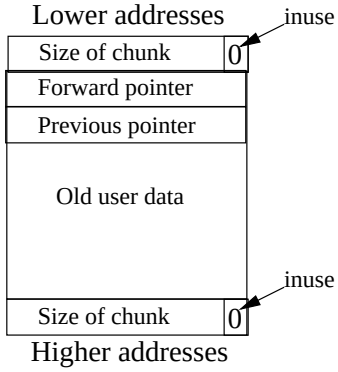

Figure 9: Free Quickfit chunk

Figure 10: Used and free Quickfit chunks

Free chunks are placed in circular doubly linked lists based on size (called bins) that are stored in the forward and backward pointers of the free chunks. These lists are accessed by an array that contains pointers to the bins. Chunks are placed in a particular bin if its size is smaller or equal to a particular bin size. Bin sizes are expressed as powers of two, but divided into four equally large intervals. When a chunk is freed it is placed at the start of its bin.

3.3.2 Exploitation

Quickfit will, like CSRI, check to see if a chunk is free or not before freeing it. As such a standard double free vulnerability is not exploitable.

Because the chunk structure of the Quickfit memory allocator is almost exactly the same as the one used in dlmalloc, the attack techniques for both the standard heap-based buffer overflow and off by one and five errors are the same. Details for this attack can be found in section 3.1.2

3.4 Poul-Henning Kamp's memory allocator

Poul-Henning Kamp's memory allocator (phkmalloc) [12, 13] is the standard memory allocator for BSD systems. It was designed for the FreeBSD operating system but has since been ported and is used on other BSD systems as well. It is different from most memory allocators in that it takes advantage of the virtual memory system present in modern architectures. It tries to minimize the pages that are accessed. One of the ways it does this is by making sure that objects smaller than or equal to a page do not span two pages. The description in this section is based on version 1.90 of phkmalloc.

3.4.1 Description

Phkmalloc relies on two layers of operation to do memory management: one layer deals with virtual pages and manages the information about the pages that have been allocated in the heap region of the program, the other layer deals with the chunks that all other memory allocators deal with. These chunks are divided into three different sizes: small, medium (up to half a page in size) and large (larger than half a page). To avoid making chunks span multiple pages, a page will only contain one large chunk. Medium and small-sized chunks will be grouped on pages that contain chunks of the same size.

The page layer The page layer manages the allocation and freeing of pages on the heap. It stores the information about all virtual pages allocated in the heap-region in a page directory. The first elements of this directory contain pointers to a linked list of pages containing chunks of each possible size. The other elements of this directory store information about particular pages: MALLOC_NOT_MINE denotes that the page was not allocated by the allocator and should be ignored, MALLOC_FREE represents the fact that the page is free. MALLOC_FIRST and MALLOC_FOLLOW are used for large chunks, they respectively mean that the page contains the first and following parts of a multipage object. Figure 11 depicts the layout of the page directory.

Pages that are free are stored in a sorted (based on address) doubly linked list of *pgfree* elements using the *next* and *prev* pointers.

```
struct pgfree {
    struct pgfree   *next;
    struct pgfree   *prev;
    void            *page;
    void            *end;
    size_t          size;
```

Adjacent free pages are coalesced: a pointer to the first pointer is kept in *page* along with the size (a multiple of pagesize) of the total of adjacent free pages. A pointer to the last free adjacent page is stored in *end*. The elements of the doubly linked list are not stored in the pages themselves: they are stored

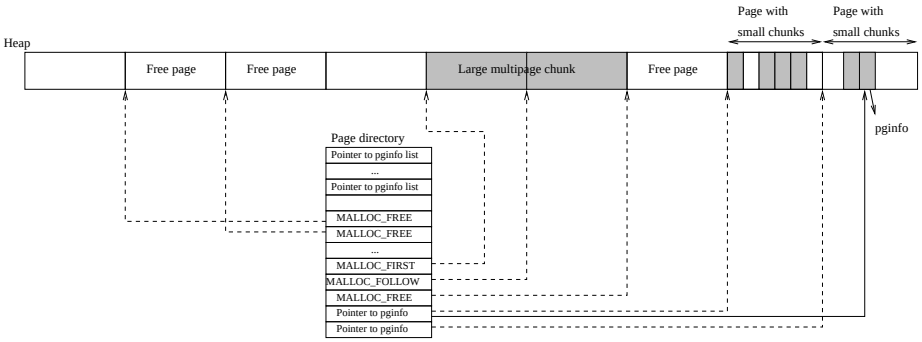

Figure 11: Page directory

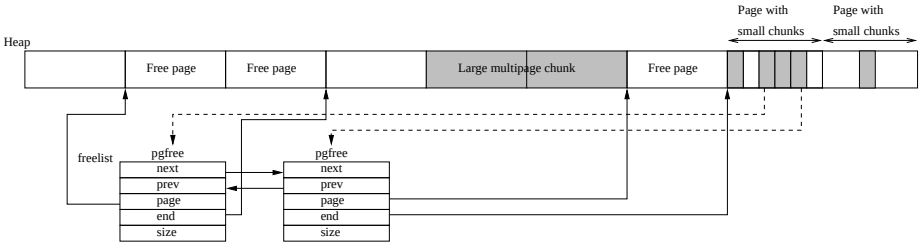

Figure 12: List of free pages

in separate datastructures, this allows the kernel to swap out the free pages because they are not accessed. As such memory must be allocated for each *pgfree* element. To increase performance, a cache pointer is kept. If a *pgfree* structure is no longer needed and the cache pointer is empty it will be set to point to the new *pgfree*, thus saving a deallocation and reallocation at a later point in time. Figure 12 illustrates what the list of free pages looks like in memory.

The chunk layer Phkmalloc distinguishes between 3 different types of chunks: small, medium and large chunks.

A large chunk which is larger than the size of half a page. When such a large chunk is allocated, the allocator will search for contiguous area of memory in the list of free pages. It will set the elements stored in the page directory to reflect the amount of pages that were allocated for this page by setting MALLOC_FIRST on the information for the element associated with the first page and MALLOC_FOLLOW for the elements of the subsequent pages (see Figure 11.

When chunks of small or medium size are requested, their size is rounded up to the next power of two. A a particular page will only contain chunks of equal sizes. A data structure, called *pginfo*, describing information about these pages is associated with every page. Depending on the size of the chunk. For small chunks *pginfo* will be stored at the beginning of its page. For medium chunks memory is allocated for the *pginfo* structure. The corresponding entry in the

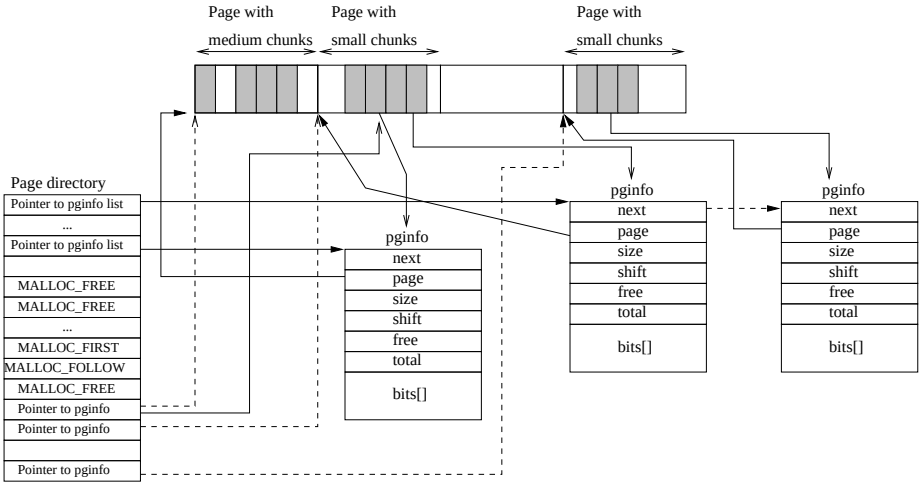

Figure 13: List of free pages

page directory for the page contain small or medium chunks is replaced with a pointer to the page's pginfo (Figure 11).

```
struct pginfo {
  struct pginfo *next;
  void          *page;
  u_short        size;
  u_short        shift;
  u_short        free;
  u_short        total;
  u_int          bits[];
```

Multiple pages of the same size are stored in a sorted linked list (based on size) that starts at page directory and is contained in the *next* pointers of each page's *pginfo*. The page pointer points to the beginning of the actual page. Size contains the size of the chunks stored on this page, while shift is used to provide a mapping from the bits array to a chunk, by bit shifting the bit's position by the amount stored in shift. Free holds the amount of free chunks that are stored on this page and total contains the total amount of chunks on this page. Bits is an array of variable size that denotes which chunks on the page are free and which are in use. Every chunk has a unique bit in the array and if the value is 1 the chunk is free.

In Figure 13 we demonstrate what the heap looks like with two small sized chunks and one medium sized chunk. The pginfo structures are allocated like normal chunks. The pginfos for a particular size are accessed via the first elements of the page directory. These elements contain a pointer to the linked list of pginfos.

3.4.2 Exploitation

Exploiting a double free is in general not possible in *phkmalloc* because the *free* function will verify if the associated bit for the chunk is set to 0 for small

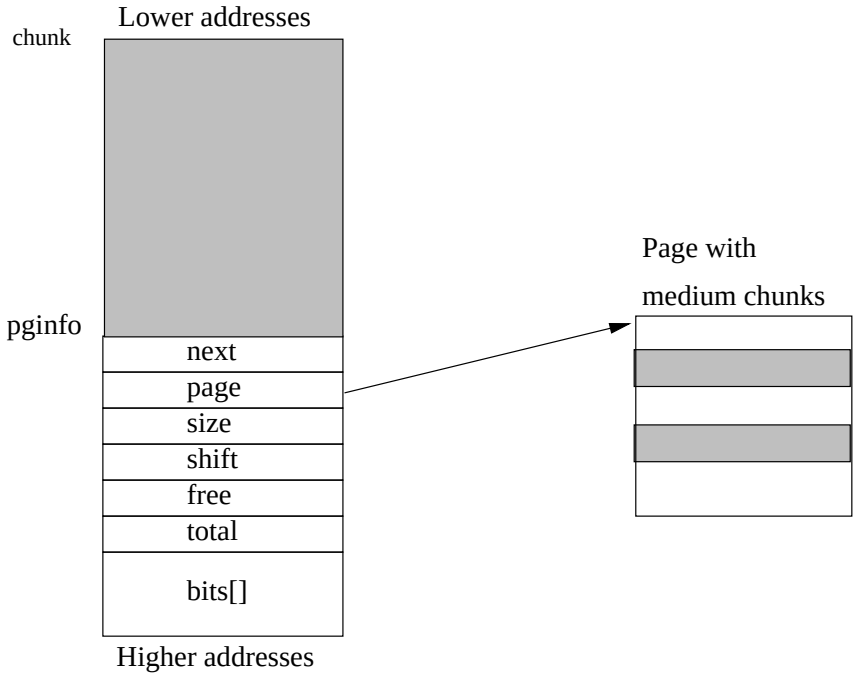

Figure 14: Overflowable chunk stored before pginfo

and medium chunks and will check if MALLOC_FREE is set in the page directory for large chunks. However because the *pginfo* structures for medium sized chunks and the *pgfree* structures are stored in memory chunks they can be overwritten by a heap overflow [3]. In this section we show how overwriting these datastructures could allow an attacker to perform a code injection attack. We start by describing an attack that overwrites pginfo and then discuss how an attacker could still perform a code-injection-attack when only one byte of adjacent memory can be overwritten.

Overwriting memory management information If a chunk stored before a *pginfo* structure is overflowable, it can overflow the information stored in *pginfo* (Figure 14).

Attackers can modify the pointer in page to point to memory they wish to overwrite and modify the bits array to make all chunks on that page seem free (Figure 15). When the allocator would want to allocate a chunk in that particular page it will return the first chunk, which is at the beginning of what page points to. As such the allocator will return the value stored in page as a valid allocated chunk. The attackers can subsequently overwrite the memory that the page has been pointed to and cause injected code to be executed.

As pgfree chunks are also stored next to normal chunks, they can also be overflowed and exploited in a similar way. For more information the reader is directed to [3]

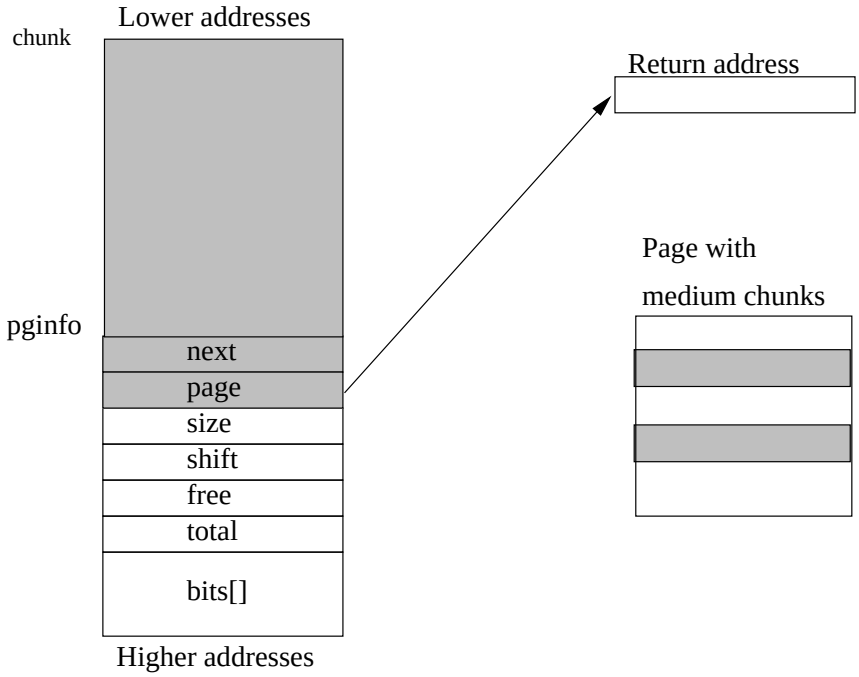

Figure 15: Overflowed chunk, modifying pginfo

Off by one If attackers can only overwrite one byte of an adjacent chunk, then they still may be able to perform a code injection attack. Instead of overwriting the page and bits field, the next field is modified (Figure 16 illustrates what the pginfo linked list looks like). The attacker makes a fake *pginfo* and then modifies the least significant bytes of the next pointer in the overflowed *pginfo* so it points to this fake structure (Figure 17). The fake *pginfo* contains the same information as the original exploit: the page pointer refers to the target memory location and the bits array denotes free chunks. When the allocator tries to allocate memory from the fake page, it will again return the target memory location as a valid chunk and thus allow the attacker to modify it.

3.5 Boehm garbage collector

The Boehm garbage collector [4–6] is a conservative garbage collector [4] for C and C++ that can be used instead of *malloc* or *new*. Programmers can request memory without having to explicitly free it when they no longer need it. The garbage collector will automatically release memory to the system when it is no longer needed. If the programmer does not interfere with memory that is managed by the garbage collector (explicit deallocation is still possible), dangling pointer references are made impossible.

[4] A conservative collector assumes that all memory locations is a pointer to another object if they contain a value that is equal to the address of an allocated chunk of memory. This can result in false negatives where some memory is incorrectly identified as still being allocated

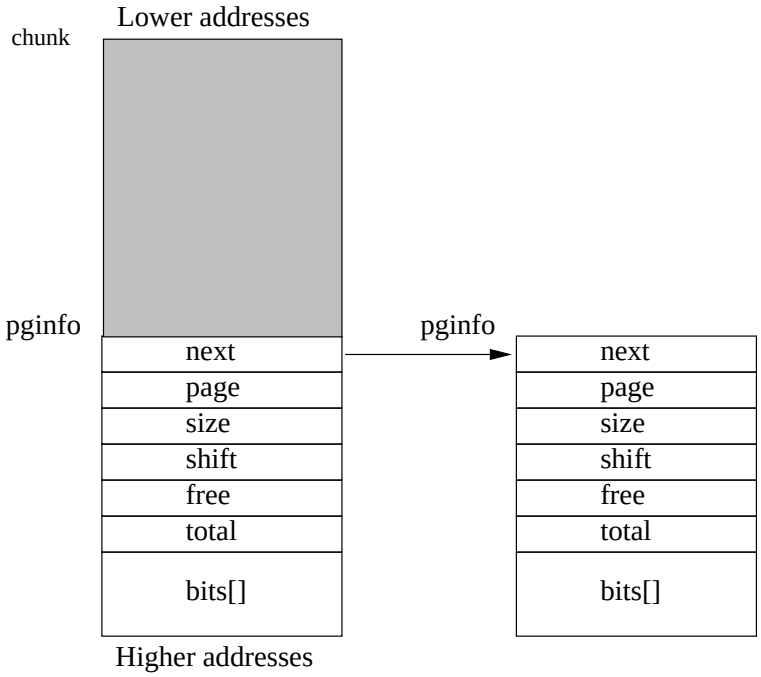

Figure 16: Pginfo linked list

3.5.1 Description

Memory is allocated by the programmer by a call to *GC_malloc* with a request for a number of bytes to allocate. The programmer can also explicitly free memory using *GC_free* or can resize the chunk by using *GC_realloc*, these two calls could however lead to dangling pointer references.

Memory structure The collector makes a difference between large and small chunks. Large chunks are larger than half of the value of HBLKSIZE[5]. These large chunks are rounded up to the next multiple of HBLKSIZE and allocated. When a small chunk is requested and none are free, the allocator will request HBLKSIZE memory from the system and divide it in small chunks of the requested size.

There is no special structure for an allocated chunk, it only contains data. A free chunk contains a pointer at the beginning of the chunk that points to the next free chunk to form a linked list of free chunks of a particular size.

Collection modes The garbage collector has two modes: incremental and non-incremental modes. In incremental mode, the heap will be increased in size whenever insufficient space is available to fulfill an allocation request. Garbage collection only starts when a certain threshold of heap size is reached. In non-incremental mode whenever a memory allocation would fail without resizing the

[5]HBLKSIZE is equal to page size on IA32.

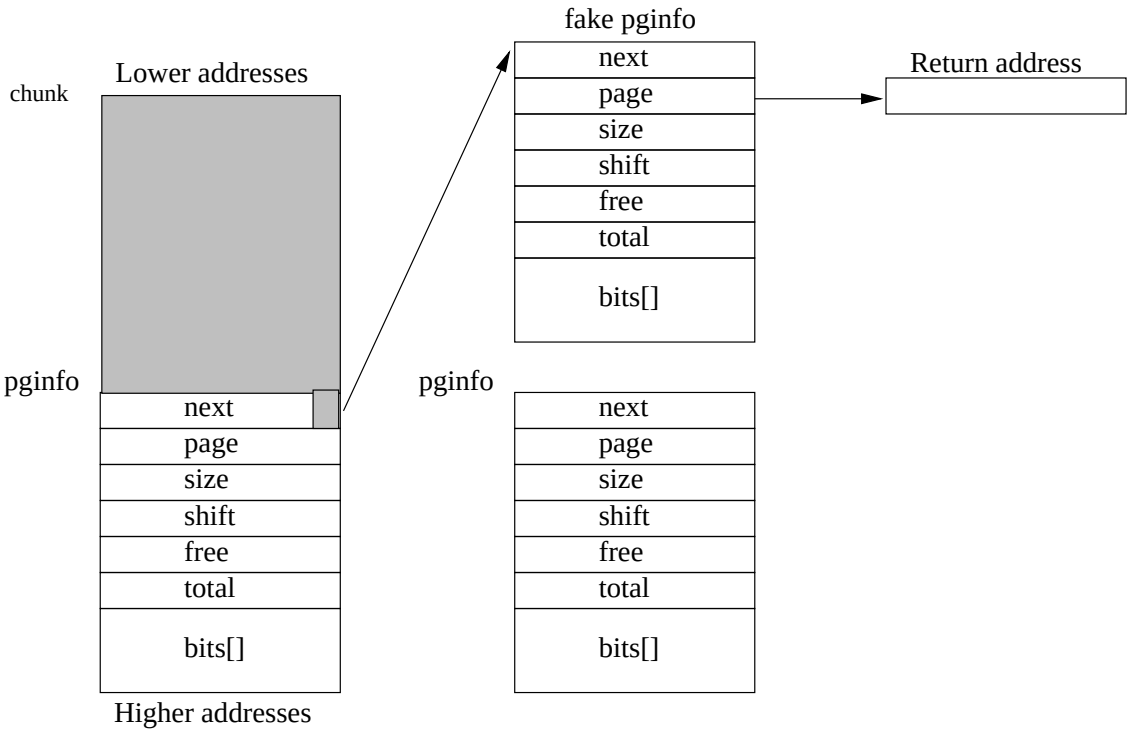

Figure 17: Exploited off by one error

heap the garbage collector decides (based on a threshold value) whether or not to start collecting.

Collection Collection is done using a mark and sweep algorithm. This algorithm works in three steps. First all objects are marked as being unreachable (i.e. candidate to be freed). The allocator then starts at the roots (registers, stack, static data) and iterates over every pointer that is reachable starting from one of these objects. When an object is reachable it is marked accordingly. Afterwards the removal phase starts: large unreachable chunks are placed in a linked list and large adjacent chunks are coalesced. Pages containing small chunks are also examined: if all of the chunks on the page are unreachable, the entire page is placed in the list of large chunks. If it is not free, the small chunks are placed in a linked list of small chunks of the same size.

3.5.2 Exploitation

Although the garbage collector removes vulnerabilities like dangling pointer references, it is still vulnerable to buffer overflows. It is also vulnerable to a double free vulnerability if the programmer explicitly frees memory.

Overwriting memory management information During the removal phase, objects are placed in a linked list of free chunks of the same size that is stored at the start of the chunk. If attackers can write out of the boundaries of a chunk, they can overwrite the pointer to the next chunk in the linked list and make it refer to the target memory location. When the allocator tries to reallocate a chunk of the same size it will return the memory location as a chunk and as a result will allow the attacker to overwrite the target memory location.

Off by one/five The vulnerability described in the previous paragraph is always an off by four vulnerability, since the attacker only needs to overwrite the first four bytes of the next chunk. If the target memory location is located close to a chunk and only the least significant byte of the pointer needs to be modified then an off by one might suffice.

Double free Dangling pointers references can not exist if the programmer does not interfere with the garbage collector. However if the programmer explicitly frees memory, a double free can occur and could be exploitable.

Figures 18 and 19 illustrate how this vulnerability can be exploited: *chunk1* was the last chunk freed and was added to the start of the linked list and points to *chunk2*. If *chunk2* is freed a second time it will be placed at the beginning of the list, but *chunk1* will still point to it. When *chunk2* is subsequently reallocated, it will be writable and still be located in the list of free chunks. The attacker can now modify the pointer and if more chunks of the same size are allocated eventually the chunk to which *chunk2* points will be returned as a valid chunk, allowing the attacker to overwrite arbitrary memory locations.

3.6 Summary

The allocators we presented in this section are just a subset of the many memory allocators that are in common use today. There are many others like the

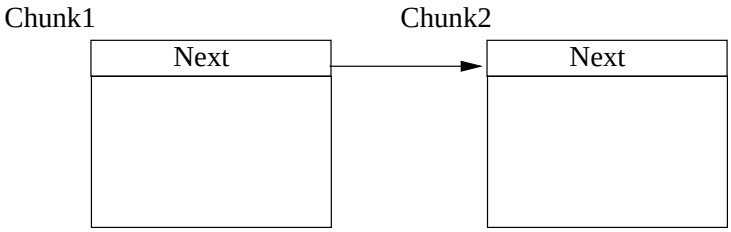

Figure 18: Linked list of free chunks

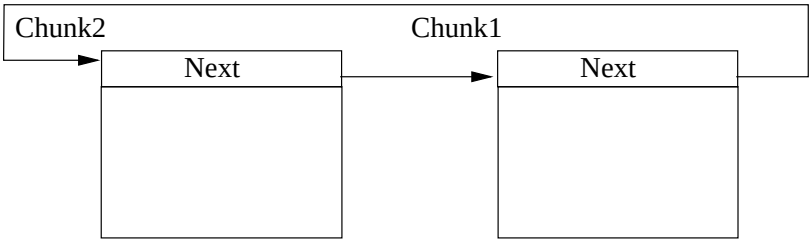

Figure 19: Double free of chunk2

memory allocator used by Windows or the allocator used in the Solaris and IRIX operating systems that are also vulnerable to similar attacks [2, 14].

4 A more secure memory allocator

As can be noted from the previous sections many allocators are vulnerable to code-injection-attacks if an attacker can modify its management information. In this section we describe a new approach to handling the management information that is more robust against these kind of attacks. This new approach could be applied to the allocators described above and we also describe a prototype implementation where we modified dlmalloc to incorporate the changes we described.

4.1 Design

On most architectures code and data are loaded into separate segments of memory and have different properties (e.g. the code segment is typically read-only, the data segment is in some cases, on architectures that support it, non-executable). We can protect the heap from code injection attacks by using a similar approach. If a chunk's chunk memory management information is separated from the data stored in the chunk i.e. if the management information is stored in separate contiguous memory areas instead of next to the chunk. We can then also protect this memory from being overwritten by an overflow by placing a non-writable page between the heap and the chunk information.

To perform the separation of the management information from the actual chunks, we use a hashtable. We map an area of memory at the end of memory (right before the stack) that is large enough to hold the hashtable. The hashtable contains pointers to a linked list of chunk information which is accessed through the hashfunction.

Figure 20 shows the memory of a process with memory regions reserved for the hashtable and the chunk information.

4.2 Prototype

We implemented the design described in the previous section by modifying dlmalloc to incorporate the changes. In this section we describe the specific details related to the implementation of our new memory allocator that we call DistriNet-malloc or dnmalloc for short.

4.2.1 Hashtable and hashfunction

To look up the memory management information associated with a chunk, the correct entry in the hashtable must be found. To do this we use the following technique: each page is divided into 256 chunks of 16 bytes (as 16 bytes is the smallest allocatable chunk). These 256 chunks are further divided into 32 groups of 8 blocks. For every group there is an entry in the hashtable: resulting in a maximum of 32 entries in the hashtable for every page. These 8 blocks are linked together through a linked list of chunks.

We find a hashentry of a particular group from a chunk's address in two steps:

1. We substract the address of the start of the heap from the chunk's address.
2. Then we shift the resulting value 7 bits to the right. This will give us the entry of the chunk's group in the hashtable.

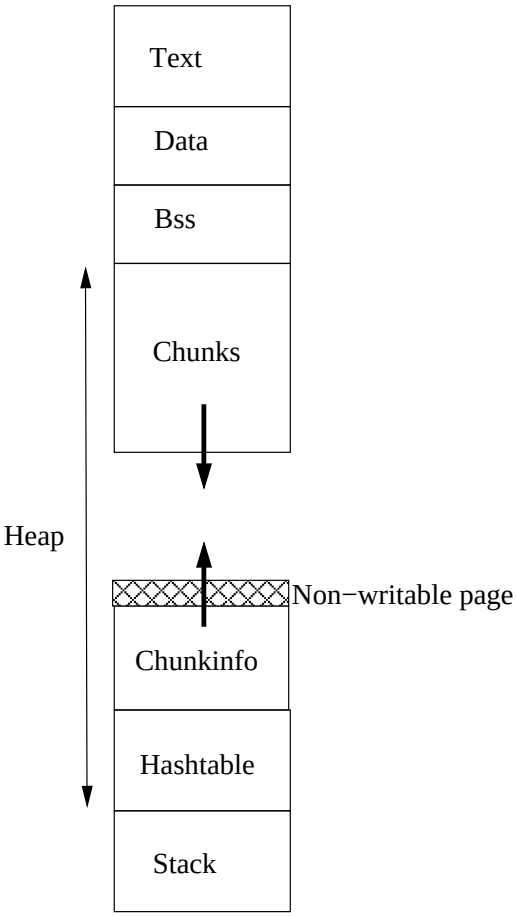

Figure 20: Modified process memory layout

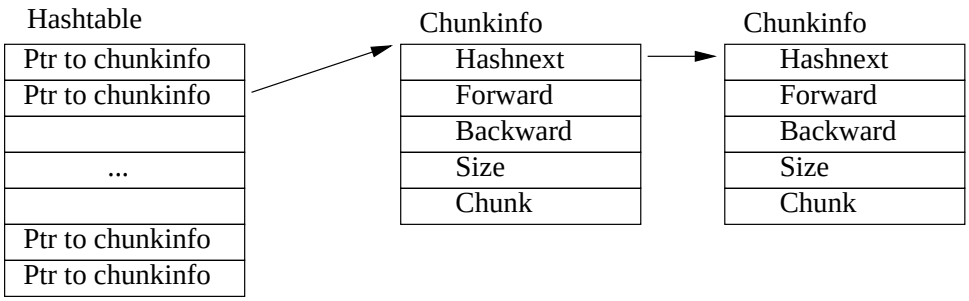

Figure 21: Hashtable and chunkinfo layout in dnmalloc

To find the chunk information associated with a chunk we now have to go over a linked list that contains a maximum of 8 entries and compare the chunk's address with the pointer to the chunk that is stored in the chunk information.

The hashtable layout is depicted in Figure 21.

4.2.2 Chunk information

This memory allocator is based on dlmalloc and besides moving the chunk information out of the chunk itself the rest of the allocator has remained the same. So we need the same information as the original allocator. Figure 21 contains the layout of our chunk information. The hashnext pointer is used to point to the next chunkinfo in group's linked list. The forward and backward pointers are used, as they are used in dlmalloc, to denote the next and previous chunks in a doubly linked list of free chunks. Size contains the size of the chunk, the three least significant bits contain the prev_inuse bit (denoting if the previous chunk is free or not), the mmapped flag (to denote if the area the chunk is stored in was allocated using *mmap* rather than *sbrk*) and a flag denoting if the chunk is in use or not. Finally, *chunk* points to the actual chunk associated with this chunkinfo.

4.2.3 Managing chunk information

The chunk information itself is stored below the hashtable. A fixed area of memory is mapped that can contain several chunkinfos. Before this area a write-protected page is mapped, to prevent the heap from overflowing into this memory region. Whenever a new chunkinfo is needed, we just allocate the next 20 bytes in the map for the chunkinfo. When we run out of space a new area is mapped below the current chunkinfos and the write protected page is moved.

Because chunks are coalesced or released back to the system we can end up with free chunkinfos. These are stored in a linked list of free chunkinfos. Before allocating the next 20 bytes of space in the map, the allocator will check the free list. If the list contains any elements then the first one will be unlinked from the list and will be used.

5 Performance overhead

Doing these extra modifications comes at a cost: both to performance and to memory overhead. To evaluate how high the overhead of dnmalloc is compared to the original dlmalloc we ran the SPEC®CPU2000 Integer reportable benchmark . Table 1 holds a description of the programs that make up the benchmark.

Program	Description
164.gzip	Data compression utility
175.vpr	FPGA circuit placement and routing
176.gcc	C compiler
181.mcf	Minimum cost network flow solver
186.crafty	Chess program
197.parser	Natural language processing
252.eon	Ray tracing
253.perlbmk	Perl
254.gap	Computational group theory
255.vortex	Object Oriented Database
256.bzip2	Data compression utility
300.twolf	Place and route simulator

Table 1: Programs used in the evaluations

This benchmark was run multiple times on 8 identical pcs (Pentium 4 2.80 Ghz, 512MB RAM, no hyperthreading, Redhat 6.2) for a total of 104 runs for each allocator. The average of the results of the benchmark for dlmalloc and dnmalloc can be found in table 2

Program	dlmalloc	dnmalloc	Overhead
gzip	253 ± 0	255.98 ± 0.01	1.18%
vpr	360.93 ± 0.16	360.55 ± 0.13	-0.11%
gcc	153.93 ± 0.05	154.76 ± 0.04	0.54%
mcf	287.19 ± 0.07	290.09 ± 0.07	1.01%
crafty	253 ± 0	254 ± 0	0.40%
parser	346.95 ± 0.02	346.61 ± 0.05	-0.10%
eon	771.05 ± 0.13	766.55 ± 0.11	-0.58%
perlbmk	243.20 ± 0.04	253.51 ± 0.05	4.24%
gap	184.07 ± 0.02	184 ± 0	-0.04%
vortex	250 ± 0	258.79 ± 0.04	3.52%
bzip2	361.64 ± 0.05	363.26 ± 0.07	0.45%
twolf	522.48 ± 0.43	513.27 ± 0.41	-1.76%

Table 2: Average SPECint®2000 results of dlmalloc and dnmalloc

The results in these tables show that the performance overhead for implementing our approach is negligible. In some cases, a program linked with our allocator outperforms the program linked with the original allocator (this is the case with vpr, parser, eon, gap and twolf). However these performance increases are so small for all programs except twolf that they can be ignored.

To examine the speed difference with twolf we executed a library trace on the program and took out all calls to malloc/free/calloc and realloc. This resulted in a trace with 561487 calls to malloc, 492709 to free, 13062 to calloc and 4 to realloc. This trace was then converted into a program and was then compared in speed by first running it with dlmalloc and subsequently with dnmalloc. The result was that the program was slower with dnmalloc. This could mean that the original program is faster with our malloc due to the difference in memory regions that are returned during allocation, which could allow for better caching for the other operations that the program performs.

6 Related work

Some work has been done on protecting the heap from overflows. In this section we examine some of the other countermeasures that have been developed to protect the management information from attack.

There are two types of allocators that try to detect or prevent heap overflow vulnerabilities: debugging allocators and runtime allocators. Debugging allocators are allocators that are meant to be used by the programmer. They can perform extra checks before using the management information stored in the chunks or ensure that the chunk is allocated in such a way that it will cause an error if it is overflowed or freed twice. Runtime allocators are meant to be used in final programs and try to protect memory allocators by performing lightweight checks to ensure that chunk information has not been modified by an attacker.

6.1 Debugging memory allocators

Both dlmalloc and CSRI have a debugging mode which will detect modification of the memory management information. When run in debug mode the allocator will check to make sure that the next pointer of the previous chunk equals the current chunk and that the previous pointer of the next chunk equals the current chunk. To exploit a heap overflow or a double free vulnerability, the pointers to the previous chunk and the next chunk must be changed.

Electric Fence [19] is a debugging library that replaces the default implementation of the malloc library with its own version that changes the behavior: every time a portion of memory is allocated it places an inaccessible virtual memory page either immediately preceding the allocated memory or immediately following it. This results in an immediate error if an under- or overrun occurs. When memory is freed, the page it is contained in will be marked inaccessible to detect dangling pointer references. This means that memory is never really freed and that the program will use two memory pages for every memory allocation.

6.2 Runtime allocators

Robertson et al. [20] designed a countermeasure that attempts to protect against attacks on the dlmalloc library management information. This is done by changing the layout of both allocated and unallocated memory chunks. To protect the management information a checksum and padding (as chunks must be of double word length) is added to every chunk. The checksum is a checksum of the management information encrypted (XOR) with a global read-only random value, to prevent attackers from generating their own checksum. When a chunk is allocated the checksum is added and when it is freed the checksum is verified. Thus if an attacker overwrites this management information with a buffer overflow a subsequent free of this chunk will abort the program because the checksum is invalid. However, this countermeasure can be bypassed if a memory leak exists in the program that would allow the attacker to print out the encryption key. The attacker can then modify the chunk information and calculate the correct value of the checksum. The allocator would then be unable to detect that the chunk information has been changed by an attacker.

Dlmalloc 2.8.x also contains extra checks to prevent the allocator from writing into memory that lies below the heap (this however doe not stop it from writing into memory that l ies above the heap (e.g. the stack)). It also offers a slightly modified version of the Robertson countermeasure as a compile-time option.

ContraPolice [15] also attempts to protect memory allocated on the heap from buffer overflows that would overwrite memory management information associated with a chunk of allocated memory. It uses the same technique as proposed by StackGuard [8], i.e. canaries to protect these memory regions. It places a randomly generated canary both before and after the memory region that it protects. Before exiting from a string or memory copying function a check is done to ensure that, if the destination region was on the heap, the canary stored before the region matches the canary stored after the region. If it does not the program is aborted. Again this countermeasure could be bypassed if the canary value could be printed out: the attacker could write past the canary and make sure to replace the canary with the same value it held before.

7 Synthesis

This section presents a summary of security vulnerabilities of the various allocators. Table 3 describes whether the memory management information can be overwritten and used to cause a code injection attack by using a heap-overflow, an off by one or a double free.

Allocator	Heap-overflow	Off by one/five	Double Free
Dlmalloc	Yes	Yes	Yes
CSRI	Yes	Yes	No
Quickfit	Yes	Yes	No
Phkmalloc	Yes	Yes	No
Boehm GC	Yes	Yes	Possibly[6]
dlmalloc (debug mode)	No	No	No
Contrapolice	No[7]	No	No
Robertson	No[7]	No	No
Dlmalloc2.8.2 (default)	Yes	Yes	No
Dnmalloc	No	No	No

Table 3: Security of different memory allocators

[6] This attack is only possible if the programmer explicitly frees memory
[7] If no memory leaks exist

8 Future work and Conclusion

One limitation in our approach, that also applies to the countermeasures described in section 6, is that it does not protect pointers stored on the heap that do not belong to the allocator. This would still allow attackers to either overwrite a function pointer stored on the heap or allow them to perform an indirect pointer overwrite on a data pointer. This is an important limitation and we will be addressing this in the future.

In this paper we examined the security of several memory allocators. We discussed how they could be exploited and showed that most memory allocators are vulnerable to code injection attacks. We then described a countermeasure that we designed that makes the memory allocator more secure against code injection attacks and showed that the overhead of using this memory allocator is negligible on program performance and memory usage.

References

[1] Aleph One. Smashing the stack for fun and profit. *Phrack*, 49, 1996.

[2] anonymous. Once upon a free(). *Phrack*, 57, 2001.

[3] BBP. BSD heap smashing. `http://www.security-protocols.com/modules.php?name=News&file=article&si%d=1586`, May 2003.

[4] Hans Boehm. Conservative gc algroithmic overview. `http://www.hpl.hp.com/personal/Hans_Boehm/gc/gcdescr.html`.

[5] Hans Boehm. A garbage collector for c and c++. `http://www.hpl.hp.com/personal/Hans_Boehm/gc/`.

[6] Hans Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software, Practice and Experience*, 18(9):807–820, September 1988.

[7] Bulba and Kil3r. Bypassing Stackguard and stackshield. *Phrack*, 56, 2000.

[8] Crispin Cowan, Calton Pu, Dave Maier, Heather Hinton, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Symposium*, pages 63–78, San Antonio, Texas, U.S.A., January 1998. USENIX Association.

[9] Igor Dobrovitski. Exploit for CVS double free() for linux pserver. `http://seclists.org/lists/bugtraq/2003/Feb/0042.html`, February 2003.

[10] Dirk Grunwald. Quickfit implementation. `ftp://ftp.cs.colorado.edu/pub/cs/misc/qf.c`.

[11] Michel Kaempf. Vudo - an object superstitiously believed to embody magical powers. *Phrack*, 57, 2001.

[12] Poul-Henning Kamp. Malloc(3) revisted. `http://phk.freebsd.dk/pubs/malloc.pdf`.

[13] Poul-Henning Kamp. Malloc implementation. `http://www.freebsd.org/cgi/cvsweb.cgi/src/lib/libc/stdlib/malloc.c`, February 2005. version 1.90.

[14] Jack Koziol, David Litchfield, Dave Aitel, Chris Anley, Sinan Eren, Neel Mehta, and Riley Hassell. *The Shellcoder's Handbook : Discovering and Exploiting Security Holes*. John Wiley & Sons, March 2004.

[15] Andreas Krennmair. ContraPolice: a libc extension for protecting applications from heap-smashing attacks. `http://www.synflood.at/contrapolice/`, November 2003.

[16] Doug Lea and Wolfram Gloger. malloc-2.7.2.c. Comments in source code.

[17] Doug Lea and Wolfram Gloger. A memory allocator. http://gee.cs.oswego.edu/dl/html/malloc.html.

[18] Mark Moraes. Csri memory allocator. `ftp://ftp.cs.toronto.edu/pub/moraes/malloc-1.18.tar.gz`, 1995.

[19] Bruce Perens. Electric fence 2.0.5. `http://perens.com/FreeSoftware/`.

[20] William Robertson, Christopher Kruegel, Darren Mutz, and Frederik Valeur. Run-time detection of heap-based overflows. In *Proceedings of the 17th Large Installation Systems Administrators Conference*, pages 51–60, San Diego, California, U.S.A., October 2003. USENIX Association.

[21] Solar Designer. JPEG COM marker processing vulnerability in netscape browsers. `http://www.openwall.com/advisories/OW-002-netscape-jpeg.txt`, July 2000.

[22] Charles B. Weinstock and William A. Wulf. Quickfit: An efficient algorithm for heap storage allocation. *ACM SIGPLAN Notices*, 23(10):141–148, October 1988.

[23] Yves Younan. An overview of common programming security vulnerabilities and possible solutions. Master's thesis, Vrije Universiteit Brussel, 2003.

[24] Yves Younan, Wouter Joosen, and Frank Piessens. Code injection in C and C++ : A survey of vulnerabilities and countermeasures. Technical Report CW386, Departement Computerwetenschappen, Katholieke Universiteit Leuven, July 2004.