Improving memory management security for C and C++

Yves Younan, Wouter Joosen, Frank Piessens, Hans Van den Eynden
DistriNet, Katholieke Universiteit Leuven, Belgium

Abstract

Memory managers are an important part of any modern language: they are used to dynamically allocate memory for use in the program. Many managers exist and depending on the operating system and language. However, two major types of managers can be identified: manual memory allocators and garbage collectors. In the case of manual memory allocators, the programmer must manually release memory back to the system when it is no longer needed. Problems can occur when a programmer forgets to release it (memory leaks), releases it twice or keeps using freed memory. These problems are solved in garbage collectors. However, both manual memory allocators and garbage collectors store management information for the memory they manage. Often, this management information is stored where a buffer overflow could allow an attacker to overwrite this information, providing a reliable way to achieve code execution when exploiting these vulnerabilities. In this paper we describe several vulnerabilities for C and C++ and how these could be exploited by modifying the management information of a representative manual memory allocator and a garbage collector.

Afterwards, we present an approach that, when applied to memory managers, will protect against these attack vectors. We implemented our approach by modifying an existing widely used memory allocator. Benchmarks show that this implementation has a negligible, sometimes even beneficial, impact on performance.

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 [2].

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 manager 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 code injection attacks [58]. However most have focused 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 a commonly used memory allocator and a garbage collector for C and C++ with respect to their resilience against code injection attacks and present a significant improvement for memory managers in order to increase robustness against code injection attacks. Our prototype implementation (which we call *dnmalloc*) comes at a very modest cost in both performance and memory usage overhead.

This paper is an extended version of work described in [59] which was presented in December 2006 at the Eighth International Conference on Information and Communication Security. The paper is structured as follows: section 2 explains which vulnerabilities can exist for heap-allocated memory. Section 3 describes how both a popular memory allocator and a garbage collector 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. 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. In section 6 related work in improving security for memory allocators is discussed. Finally, section 7 discusses possible future enhancements 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. Buffer overflow, which are usually exploited on the stack, are also possible in this kind of memory. Exploitation of such heap-based buffer overflows usually relies on finding either function pointers or by performing an indirect pointer attack [17] on data pointers in this memory area. However, these pointers are not always present in the data stored by the program in this memory. 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 [4,56]. In the following sections we will describe how an attacker could use specific memory managers to perform this kind of attack.

2.2 Off by one 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 [4,56]. A more generally exploitable version of the off by one for memory allocators is an off by five, while these do not occur as 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 dereferencing a dangling pointer will lead to a program crash. However in heap memory, it could also 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 memory chunk and as a result could lead to a code injection attack [22]. This kind of vulnerability is not present in all memory managers, as some will check if a chunk is free or not before freeing it a second time. It may also be possible to write to memory which has already been reused, while the program think it is still writing to the original object. This can also lead to vulnerabilities. These last kind of vulnerabilities are, however, much harder to exploit in general programs than a double free. The possibility of exploiting these vulnerabilities will most likely rely on the way the program uses the memory rather than by using the memory manager to the attacker’s advantage.

In the following sections we will describe a specific memory allocator could be exploited using dangling pointer references and more specifically, double free vulnerabilities. More information about these attacks can be found in [22,56,60].

3 Memory managers

In this section we will examine a representative memory allocator and a garbage collector for C and C++. We have chosen Doug Lea's memory allocator on which the Linux memory allocator is based, because this allocator is in wide use and illustrates typical vulnerabilities that are encountered in other memory allocators. A discussion of how other memory allocators can be exploited by attackers can be found in [60]. Boehm's garbage collector was chosen to determine whether a representative garbage collecting memory manager for C/C++ would be more resilient against attack.

We will describe how these memory managers work in normal circumstances and then will explain how a heap-vulnerability that can overwrite the management information of these memory managers could be used by an attacker to cause a code injection attack. We will use the same structure to describe both memory managers: first we describe how the manager 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 [39,40] (commonly referred to as *dlmalloc*) was designed as a general-purpose memory allocator that could be used by any kind of program. *Dlmalloc* is used as the basis for *ptmalloc* [24], which is the allocator used in the GNU/Linux operating system. *Ptmalloc* mainly differs from dlmalloc in that it offers better support for multithreading, however this has no direct impact on the way an attacker can abuse the memory allocator's management information to perform code injection attacks. The description of *dlmalloc* 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 a doubly linked list, sorted by size. 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 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. *Chunk3* is a free chunk that is allocated adjacent to *chunk1*. *Chunk2* and *chunk4* are free chunks located in arbitrary locations on the heap.

Chunk3 is located in a doubly linked list together with *chunk2* and *chunk4*. *Chunk2* is the first chunk in the chain: its forward pointer points to *chunk3* and its backward pointer points to a previous chunk in the list. *Chunk3*'s forward pointer points to *chunk4* and its backward pointer points to *chunk2*. *Chunk4* is the last chunk in our example: its forward pointer points to a next chunk in the list and its backward pointer points to *chunk3*.

3.1.2 Exploitation

Dlmalloc is vulnerable to all three of the previously described vulnerabilities [4,22,31,48]. Here we will describe how these vulnerabilities may lead to a code injection attack.

[1]A chunk is a block of memory that is allocated by the allocator, it can be larger than what a 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 (P) or not and one to indicate if the memory is mapped or not (M). The third 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.

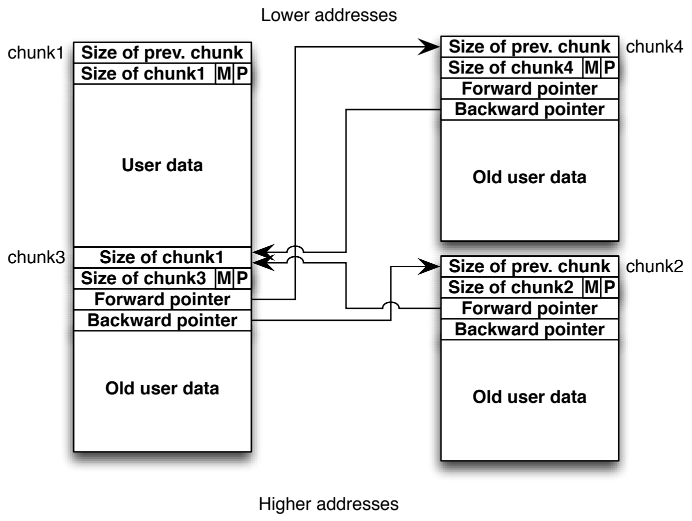

Figure 1: Heap containing used and free chunks

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 *chunk3*. 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 *chunk3* into a larger chunk. As *chunk3* 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*, *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 injected code. 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 [4, 31, 48].

Off by one error An off by one error could also be exploited in the Doug Lea's memory allocator [4]. 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 of the chunk, the least significant byte of the size field of the next chunk will be overwritten. 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

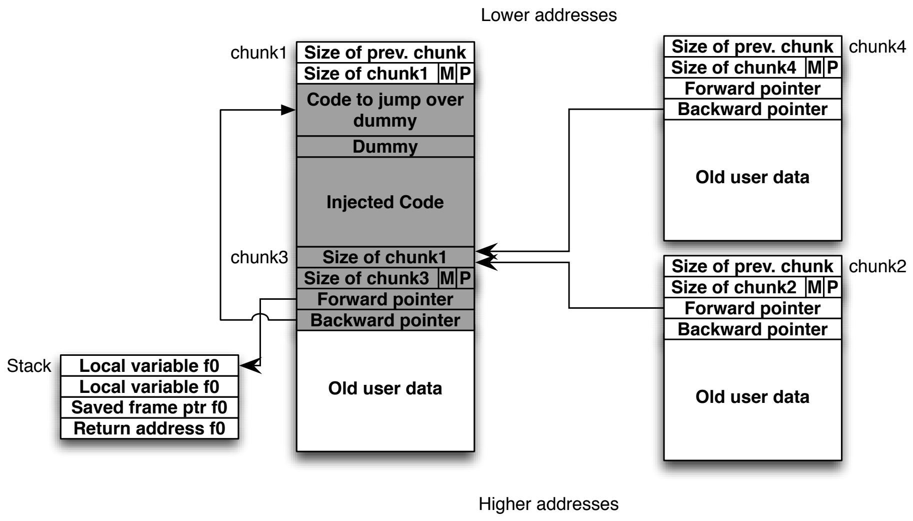

Figure 2: Heap-based buffer overflow in *dlmalloc*

the current chunk as free. The same technique using the forward and backward pointers (in the fake chunk) 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 [22]. 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 larger than *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):

```
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 in succession, 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
```

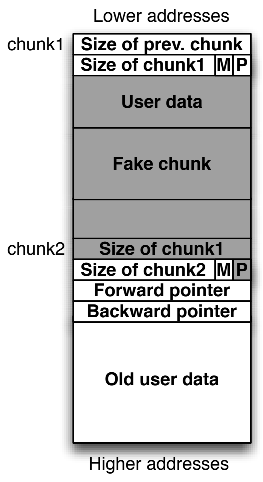

Figure 3: Off by one error in *dlmalloc*

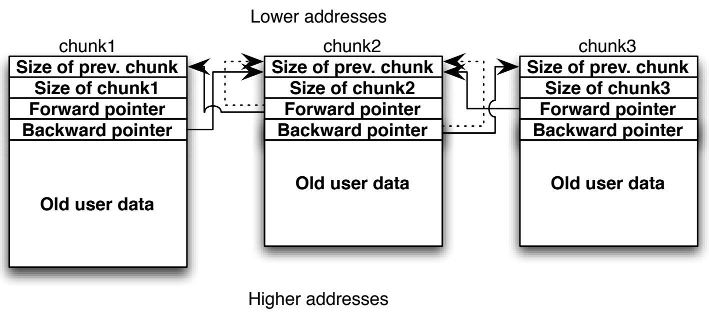

Figure 4: List of free chunks in dlmalloc: full lines show a normal list of chunks, dotted lines show the changes after a double free has occurred.

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

3.2 Boehm garbage collector

The Boehm garbage collector [14–16] is a conservative garbage collector [3] 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.

3.2.1 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[4]. 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 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. candidates 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] A conservative collector assumes that each memory location is a pointer to another object if it contains 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.

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

3.2.2 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 five The garbage collector will automatically add padding to an object to ensure that the property of C/C++ which allows a pointer to point to one element past an array is recognized as pointing to the object rather than the next. This padding forces an attacker to overwrite the padding (4 bytes on IA32). He can then overwrite the first four bytes of the next chunk with an off by eight attack. 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 five might suffice.

Double free Dangling pointer references cannot 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 5 and 6 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.

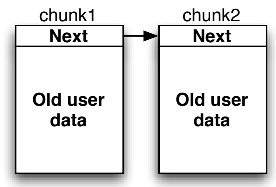

Figure 5: Linked list of free chunks in Boehm's garbage collector

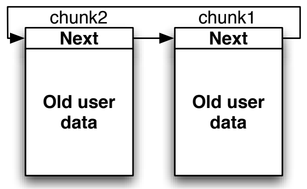

Figure 6: Double free of *chunk2* in Boehm's garbage collector

3.3 Summary

The memory allocator we presented in this section is representative for the many memory allocators that are in common use today. There are many others like the memory allocator used by Windows, the allocator used in the Solaris and IRIX operating systems or the allocator used in FreeBSD that are also vulnerable to similar attacks [4, 9, 36].

Very few garbage collectors exist for C and C++, in the previous section we also discussed how a garbage collector can be vulnerable to the same attacks that are often performed on memory allocators.

4 A more secure memory allocator

As can be noted from the previous sections many memory managers 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 managers

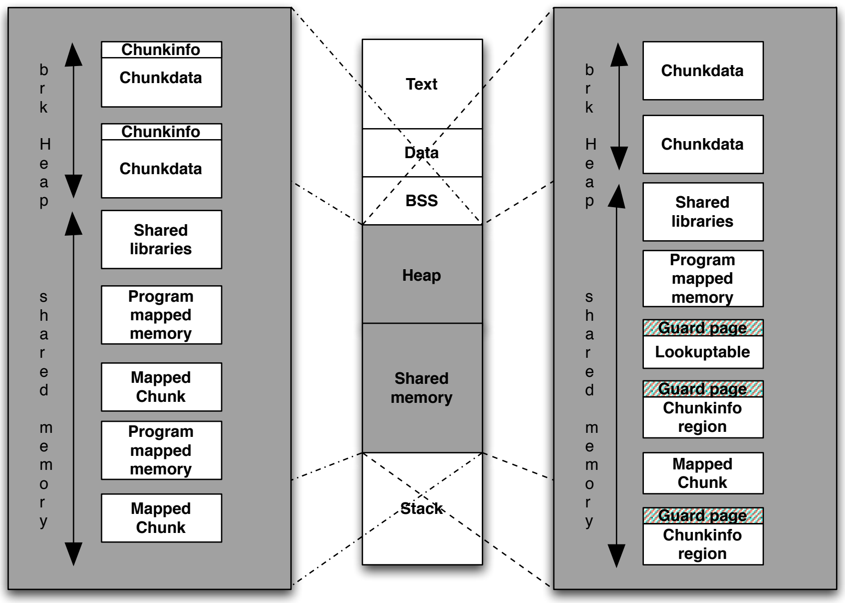

Figure 7: Original (left) and modified (right) process memory layout

discussed above and we also describe a prototype implementation (called *dnmalloc*) where we modified *dlmalloc* to incorporate the changes we described.

4.1 Countermeasure Design

The main principle used to design this countermeasure is to separate management information (*chunkinfo*) from the data stored by the user (*chunkdata*). This management information is then stored in separate contiguous memory regions that only contain other management information. To protect these regions from being overwritten by overflows in other memory mapped areas, they are protected by guard pages. This simple design essentially makes overwriting the *chunkinfo* by using a heap-based buffer overflow impossible. Figure 7 depicts the typical memory layout of a program that uses a general memory allocator (on the left) and one that uses our modified design (on the right)

Most memory allocators will allocate memory in the datasegment that could be increased (or decreased) as necessary using the *brk* systemcall [50]. However, when larger chunks are requested, it can also allocate memory in the shared memory area [5] using the *mmap*[6] systemcall to allocate memory for the chunk. In Fig. 7, we have depicted this behavior: there are chunks allocated in both the heap and in the shared memory area. Note that a program can also map files and devices into this region itself, we have depicted this in Fig. 7 in the boxes labeled *'Program mapped memory'*.

In this section we describe the structures needed to perform this separation in a memory allocator efficiently. In the next paragraph we describe the structures that are used to retrieve the *chunkinfo* when presented with a pointer to *chunkdata*. In the paragraph that follows the next, we discuss the management of the region where these *chunkinfos* are stored.

[5]Note that memory in this area is not necessarily shared among applications, it has been allocated by using *mmap*

[6]mmap is used to map files or devices into memory. However, when passing it the *MAP_ANON* flag or mapping the */dev/zero* file, it can be used to allocate a specific region of contiguous memory for use by the application (however, the granularity is restricted to page size) [50].

4.1.1 Lookup table and lookup function

To perform the separation of the management information from the actual *chunkdata*, we use a *lookup table*. The entries in the *lookup table* contain pointers to the *chunkinfo* for a particular *chunkdata*. When given such a *chunkdata* address, a lookup function is used to find the correct entry in the *lookup table*.

The table is stored in a map of contiguous memory that is big enough to hold the maximum size of the *lookup table*. This map can be large on 32-bit systems, however it will only use virtual address space rather than physical memory. Physical memory will only be allocated by the operating system when the specific page is written to. To protect this memory from buffer overflows in other memory in the shared memory region, a guard page is placed before it. At the right hand side of Fig. 7 we illustrate what the layout looks like in a typical program that uses this design.

4.1.2 Chunkinfo regions

Chunkinfos are also stored in a particular contiguous region of memory (called a *chunkinfo region*), which is protected from other memory by a guard page. This region also needs to be managed, several options are available for doing this. We will discuss the advantages and disadvantages of each.

Our preferred design, which is also the one used in our implementation and the one depicted in Fig. 7, is to map a region of memory large enough to hold a predetermined amount of *chunkinfos*. To protect its contents, we place a guard page at the top of the region. When the region is full, a new region, with its own guard page, is mapped and added to a linked list of *chunkinfo regions*. This region then becomes the active region, meaning that all requests for new *chunkinfos* that cannot be satisfied by existing *chunkinfos*, will be allocated in this region. The disadvantage of this technique is that a separate guard page is needed for every *chunkinfo region*, because the allocator or program may have stored data in the same region (as depicted in Fig. 7). Although such a guard page does not need actual memory (it will only use virtual memory), setting the correct permissions for it is an expensive system call (requiring the system to perform several time-consuming actions to execute).

When a *chunkdata* disappears, either because the associated memory is released back to the system or because two *chunkdatas* are coalesced into one, the *chunkinfo* is stored in a linked list of free *chunkinfos*. In this design, we have a separate list of free *chunkinfos* for every region. This list is contained in one of the fields of the *chunkinfo* that is unused because it is no longer associated with a *chunkdata*. When a new *chunkinfo* is needed, the allocator returns one of these free *chunkinfos*: it goes over the lists of free *chunkinfos* of all existing *chunkinfo regions* (starting at the currently active region) to attempt to find one. If none can be found, it allocates a new *chunkinfo* from the active region. If all *chunkinfos* for a region have been added to its list of free *chunkinfos*, the entire region is released back to the system.

An alternative design is to map a single *chunkinfo region* into memory large enough to hold a specific amount of *chunkinfos*. When the map is full, it can be extended as needed. The advantage is that there is one large region, and as such, not much management is required on the region, except growing and shrinking it as needed. This also means that we only need a single guard page at the top of the region to protect the entire region. However, a major disadvantage of this technique is that, if the virtual address space behind the region is not free, extension means moving it somewhere else in the address space. While the move operation is not expensive because of the paging system used in modern operating systems, it invalidates the pointers in the *lookup table*. Going over the entire *lookup table* and modifying the pointers is prohibitively expensive. A possible solution to this is to store offsets in the *lookup table* and to calculate the actual address of the *chunkinfo* based on the base address of the *chunkinfo region*.

A third design is to store the *chunkinfo region* directly below the maximum size the stack can grow to (if the stack has such a fixed maximum size), and make the *chunkinfo region* grow down toward the heap. This eliminates the problem of invalidation as well, and does not require extra calculations to find a *chunkinfo*, given an entry in the *lookup table*. To protect this region from being overwritten by data stored on the heap, a guard page has to be placed at the top of the region, and has to be moved every time the region is extended. A major disadvantage of this technique is that it can be hard to determine the start of the stack region on systems that use address space layout randomization [51]. It is also incompatible with programs that do not have a fixed maximum stack size.

These last two designs only need a single, but sorted, list of free *chunkinfos*. When a new *chunkinfo* is needed, it can return, respectively, the lowest or highest address from this list. When the free list reaches a predetermined size, the region can be shrunk and the active *chunkinfos* in the shrunk area are copied to free space in the remaining

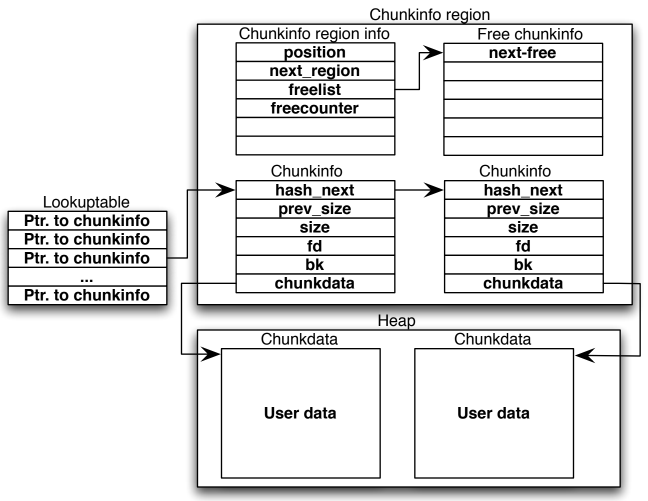

Figure 8: *Lookup table* and chunkinfo layout

chunkinfo region.

4.2 Prototype Implementation

Dnmalloc was implemented by modifying *dlmalloc 2.7.2* to incorporate the changes described in Section 4.1. The ideas used to build this implementation, however, could also be applied to other memory allocators. *Dlmalloc* was chosen because it is very widely used (in its *ptmalloc* incarnation) and is representative for this type of memory allocators. *Dlmalloc* was chosen over *ptmalloc* because it is less complex to modify and because the modifications done to *dlmalloc* to achieve *ptmalloc* do not have a direct impact on the way the memory allocator can be abused by an attacker.

4.2.1 Lookup table and lookup function

The *lookup table* is in fact a lightweight hashtable: to implement it, we divide every page in 256 possible chunks of 16 bytes (the minimum chunk size), which is the maximum amount of chunks that can be stored on a single page in the heap. These 256 possible chunks are then further divided into 32 groups of 8 elements. For every such group we have 1 entry in the *lookup table* that contains a pointer to a linked list of these elements (which has a maximum size of 8 elements). As a result we have a maximum of 32 entries for every page. The *lookup table* is allocated using the memory mapping function, mmap. This allows us to reserve virtual address space for the maximum size that the *lookup table* can become without using physical memory. Whenever a new page in the *lookup table* is accessed, the operating system will allocate physical memory for it.

We find an entry in the table for a particular group from a *chunkdata*'s address in two steps:

1. We subtract the address of the start of the heap from the *chunkdata*'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 *lookup table*.

To find the *chunkinfo* associated with a chunk we now have to go over a linked list that contains a maximum of 8 entries and compare the *chunkdata*'s address with the pointer to the *chunkdata* that is stored in the *chunkinfo*. This linked list is stored in the hashnext field of the *chunkinfo* (illustrated in Fig. 8).

4.2.2 Chunkinfo

A *chunkinfo* contains all the information that is available in *dlmalloc*, and adds several extra fields to correctly maintain the state. The layout of a *chunkinfo* is illustrated in Fig. 8: the *prev_size*, *size*, *forward* and *backward* pointers serve the same purpose as they do in *dlmalloc*, the *hashnext* field contains the linked list that we mentioned in the previous section and the *chunkdata* field contains a pointer to the actual allocated memory.

4.3 Managing chunk information

The chunk information itself is stored in a fixed map that is big enough to hold a predetermined amount of *chunkinfos*. Before this area a guard page is mapped, to prevent the heap from overflowing into this memory region. Whenever a new *chunkinfo* is needed, we simply allocate the next 24 bytes in the map for the *chunkinfo*. When we run out of space, a new region is mapped together with a guard page.

One *chunkinfo* in the region is used to store the meta-data associated with a region. This metadata (illustrated in Fig. 8, by the *chunkinfo region info* structure) contains a pointer to the start of the list of free chunks in the freelist field. It also holds a counter to determine the current amount of free *chunkinfos* in the region. When this number reaches the maximum amount of chunks that can be allocated in the region, it will be deallocated. The *chunkinfo region info* structure also contains a position field that determines where in the region to allocate the next *chunkinfo*. Finally, the next_region field contains a pointer to the next *chunkinfo* region.

5 Evaluation

The realization of these extra modifications comes at a cost: both in terms of performance and in terms of memory overhead. To evaluate how high the performance overhead of *dnmalloc* is compared to the original *dlmalloc*, we ran the full SPEC® CPU2000 Integer reportable benchmark [27], which gives us an idea of the overhead associated with general-purpose programs. We also evaluated the implementation using a suite of allocator-intensive benchmarks, which have been widely used to evaluate the performance of memory managers [11,12,26,29]. While these two suites of benchmarks make up the macrobenchmarks of this section, we also performed microbenchmarks to get a better understanding of which allocator functions are faster or slower when using *dnmalloc*.

Table 1 holds a description of the programs that were used in both the macro- and the microbenchmarks. For all the benchmarked applications we have also included the number of times they call the most important memory allocation functions: *malloc*, *realloc*, *calloc*[7] and free (the SPEC® benchmark calls programs multiple times with different inputs for a single run; for these we have taken the average number of calls).

The results of the performance evaluation can be found in Section 5.1. Both macrobenchmarks and the microbenchmarks were also used to measure the memory overhead of our prototype implementation compared to *dlmalloc*. In Section 5.2 we discuss these results. Finally, we also performed an evaluation of the security of *dnmalloc* in Section 5.3 by running a set of exploits against real world programs using both *dlmalloc* and *dnmalloc*.

Dnmalloc and all files needed to reproduce these benchmarks are available publicly [57].

5.1 Performance

This section evaluates our countermeasure in terms of performance overhead. All benchmarks were run on 10 identical machines (Pentium 4 2.80 Ghz, 512MB RAM, no hyperthreading, Redhat 6.2, kernel 2.6.8.1).

5.1.1 Macrobenchmarks

To perform these benchmarks, the SPEC® benchmark was run 10 times on these PCs for a total of 100 runs for each allocator. The allocator-intensive benchmarks were run 50 times on the 10 PCs for a total of 500 runs for each allocator.

[7]This memory allocator call will allocate memory and will then clear it by ensuring that all memory is set to 0

SPEC CPU2000 Integer benchmark programs					
Program	Description	malloc	realloc	calloc	free
164.gzip	Data compression utility	87,241	0	0	87,237
175.vpr	FPGA placement routing	53,774	9	48	51,711
176.gcc	C compiler	22,056	2	0	18,799
181.mcf	Network flow solver	2	0	3	5
186.crafty	Chess program	39	0	0	2
197.parser	Natural language processing	147	0	0	145
252.eon	Ray tracing	1,753	0	0	1,373
253.perlbmk	Perl	4,412,493	195,074	0	4,317,092
254.gap	Computational group theory	66	0	1	66
255.vortex	Object Oriented Database	6	0	1,540,780	1,467,029
256.bzip2	Data compression utility	12	0	0	2
300.twolf	Place and route simulator	561,505	4	13,062	492,727
Allocator-intensive benchmarks					
Program	Description	malloc	realloc	calloc	free
boxed-sim	Balls-in-box simulator	3,328,299	63	0	3,312,113
cfrac	Factors numbers	581,336,282	0	0	581,336,281
espresso	Optimizer for PLAs	5,084,290	59,238	0	5,084,225
lindsay	Hypercube simulator	19,257,147	0	0	19,257,147

Table 1: Programs used in the evaluations of *dnmalloc*

SPEC CPU2000 Integer benchmark programs			
Program	Dlmalloc r/t (s)	Dnmalloc r/t (s)	R/t overhead
164.gzip	253 ± 0	253 ± 0	0%
175.vpr	361 ± 0.15	361.2 ± 0.14	0.05%
176.gcc	153.9 ± 0.05	154.1 ± 0.04	0.13%
181.mcf	287.3 ± 0.07	290.1 ± 0.07	1%
186.crafty	253 ± 0	252.9 ± 0.03	-0.06%
197.parser	347 ± 0.01	347 ± 0.01	0%
252.eon	770.3 ± 0.17	782.6 ± 0.1	1.6%
253.perlbmk	243.2 ± 0.04	255 ± 0.01	4.86%
254.gap	184.1 ± 0.03	184 ± 0	-0.04%
255.vortex	250.2 ± 0.04	223.6 ± 0.05	-10.61%
256.bzip2	361.7 ± 0.05	363 ± 0.01	0.35%
300.twolf	522.9 ± 0.44	511.9 ± 0.55	-2.11%
Allocator-intensive benchmarks			
Program	Dlmalloc r/t (s)	Dnmalloc r/t (s)	R/t overhead
boxed-sim	230.6 ± 0.08	232.2 ± 0.12	0.73%
cfrac	552.9 ± 0.05	587.9 ± 0.01	6.34%
espresso	60 ± 0.02	60.3 ± 0.01	0.52%
lindsay	239.1 ± 0.02	242.3 ± 0.02	1.33%

Table 2: Average macrobenchmark runtime results for *dlmalloc* and *dnmalloc*

Microbenchmarks			
Program	DL r/t	DL r/t	R/t Overh.
loop: malloc	0.28721 ± 0.00108	0.06488 ± 0.00007	-77.41%
loop: realloc	1.99831 ± 0.00055	1.4608 ± 0.00135	-26.9%
loop: free	0.06737 ± 0.00001	0.03691 ± 0.00001	-45.21%
loop: calloc	0.32744 ± 0.00096	0.2142 ± 0.00009	-34.58%
loop2: malloc	0.32283 ± 0.00085	0.39401 ± 0.00112	22.05%
loop2: realloc	2.11842 ± 0.00076	1.26672 ± 0.00105	-40.2%
loop2: free	0.06754 ± 0.00001	0.03719 ± 0.00005	-44.94%
loop2: calloc	0.36083 ± 0.00111	0.1999 ± 0.00004	-44.6%

Table 3: Average microbenchmark runtime results for *dlmalloc* and *dnmalloc*

Table 2 contains the average runtime, including standard error, of the programs in seconds. The results show that the runtime overhead of our allocator are mostly negligible both for general programs as for allocator-intensive programs. However, for *perlbmk* and *cfrac* the performance overhead is slightly higher: 4% and 6%. These show that even for such programs the overhead for the added security is extremely low. In some cases (*vortex* and *twolf*) the allocator even improves performance. This is mainly because of improved locality of management information in our approach: in general all the management information for several chunks will be on the same page, which results in more cache hits [26]. When running the same tests on a similar system with L1 and L2 cache[8] disabled, the performance benefit for *vortex* went down from 10% to 4.5%.

5.1.2 Microbenchmarks

We have included two microbenchmarks. In the first microbenchmark, the time that the program takes to perform 100,000 *mallocs* of random[9] chunk sizes ranging between 16 and 4096 bytes was measured. Afterwards the time was measured for the same program to *realloc* these chunks to different random size (also ranging between 16 and 4096 bytes). We then measured how long it took the program to *free* those chunks and finally to *calloc* 100,000 new chunks of random sizes. The second benchmark does essentially the same but also performs a *memset*[10] on the memory it allocates (using *malloc*, *realloc* and *calloc*). The microbenchmarks were each run 100 times on a single PC (the same configuration as was used for the macrobenchmarks) for each allocator.

The average of the results (in seconds) of these benchmarks, including the standard error, for *dlmalloc* and *dnmalloc* can be found in Table 3. Although it may seem from the results of the *loop* program that the *malloc* call has an enormous speed benefit when using *dnmalloc*, this is mainly because our implementation does not access the memory it requests from the system. This means that on systems that use optimistic memory allocation (which is the default behavior on Linux) our allocator will only use memory when the program accesses it.

To measure the actual overhead of our allocator when the memory is accessed by the application, we also performed the same benchmark in the program *loop2*, but in this case always set all bytes in the acquired memory to a specific value. Again there are some caveats in the measured result: while it may seem that the *calloc* function is much faster, in fact it has the same overhead as the *malloc* function followed by a call to *memset* (because *calloc* will call *malloc* and then set all bytes in the memory to 0). However, the place where it is called in the program is of importance here: it was called after a significant amount of chunks were freed and as a result this call will reuse existing free chunks. Calling *malloc* in this case would have produced similar results.

The main conclusion we can draw from these microbenchmarks is that the performance of our implementation is very close to that of *dlmalloc*: it is faster for some operations, but slower for others.

[8]These are caches that are faster than the actual memory in a computer and are used to reduce the cost of accessing general memory [52].

[9]Although a fixed seed was set so two runs of the program return the same results

[10]This call will fill a particular range in memory with a particular byte.

5.2 Memory overhead

Our implementation also has an overhead when it comes to memory usage: the original allocator has an overhead of approximately 8 bytes per chunk. Our implementation has an overhead of approximately 24 bytes to store the chunk information and for every 8 chunks, a *lookup table* entry will be used (4 bytes). Depending on whether the chunks that the program uses are large or small, our overhead could be low or high. To test the memory overhead on real world programs, we measured the memory overhead for the benchmarks we used to test performance, the results (in megabytes) can be found in Table 4. They contain the complete overhead of all extra memory the countermeasure uses compared to *dlmalloc*.

SPEC CPU2000 Integer benchmark programs			
Program	dlmalloc mem. use (MB)	our mem. use (MB)	Overhead
164.gzip	180.37	180.37	0%
175.vpr	20.07	20.82	3.7%
176.gcc	81.02	81.14	0.16%
181.mcf	94.92	94.92	0%
186.crafty	0.84	0.84	0.12%
197.parser	30.08	30.08	0%
252.eon	0.33	0.34	4.23%
253.perlbmk	53.80	63.37	17.8%
254.gap	192.07	192.07	0%
255.vortex	60.17	63.65	5.78%
256.bzip2	184.92	184.92	0%
300.twolf	3.22	5.96	84.93%
Allocator-intensive benchmarks			
Program	dlmalloc mem. use (MB)	our mem. use (MB)	Overhead
boxed-sim	0.78	1.16	49.31%
cfrac	2.14	3.41	59.13%
espresso	5.11	5.88	15.1%
lindsay	1.52	1.57	2.86%
Microbenchmarks			
loop/loop2	213.72	217.06	1.56%

Table 4: Average memory usage for *dlmalloc* and *dnmalloc*

In general, the relative memory overhead of our countermeasure is fairly low (generally below 20%), but in some cases the relative overhead can be very high, this is the case for *twolf*, *boxed-sim* and *cfrac*. These applications use many very small chunks, so while the relative overhead may seem high, if we examine the absolute overhead it is fairly low (ranging from 120 KB to 2.8 MB). Applications that use larger chunks have a much smaller relative memory overhead.

5.3 Security evaluation

In this section we present experimental results when using our memory allocator to protect applications with known vulnerabilities against existing exploits.

Table 5 contains the results of running several exploits against known vulnerabilities when these programs were compiled using *dlmalloc* and *dnmalloc* respectively. When running the exploits against *dlmalloc*, we were able to execute a code injection attack in all cases. However, when attempting to exploit *dnmalloc*, the overflow would write into adjacent chunks, but would not overwrite the management information, as a result, the programs kept running.

Exploit for	Dlmalloc	Dnmalloc
Wu-ftpd 2.6.1 [61]	Shell	Continues
Sudo 1.6.1 [32]	Shell	Crash
Sample heap-based buffer overflow	Shell	Continues
Sample double free	Shell	Continues

Table 5: Results of exploits against vulnerable programs protected with *dnmalloc*

These kinds of security evaluations can only prove that a particular attack works, but it cannot disprove that no variation of this attack exists that does work. Because of the fragility of exploits, a simple modification in which an extra field is added to the memory management information for the program would cause many exploits to fail. While this is useful against automated attacks, it does not provide any real protection from a determined attacker. Testing exploits against a security solution can only be used to prove that it can be bypassed. As such, we provide these evaluations to demonstrate how our countermeasure performs when confronted with a real world attack, but we do not make any claims as to how accurately they evaluate the security benefit of *dnmalloc*.

However, the design in itself of the allocator gives strong security guarantees against buffer overflows, since none of the memory management information is stored with user data. We contend that it is impossible to overwrite it using a heap-based buffer overflow. If such an overflow occurs, an attacker will start at a chunk and will be able to overwrite any data that is behind it. Since such an buffer overflow is contiguous, the attacker will not be able to overwrite the management information. If an attacker is able to write until the management information, it will be protected by the guard page. An attacker could use a pointer stored in heap memory to overwrite the management information, but this would be a fairly useless operation: the management information is only used to be able to modify a more interesting memory location. If attackers already control a pointer they could overwrite the target memory location directly instead of going through an extra level of indirection.

Our approach does not *detect* when a buffer overflow has occurred. It is, however, possible to easily and efficiently add such detection as an extension to dnmalloc. A technique similar to the one used in [37, 44] could be added to the allocator by placing a random number at the top of a chunk (where the old management information used to be) and by mirroring that number in the management information. Before performing any heap operation (i.e. malloc, free, coalesce, etc) on a chunk, the numbers would be compared and if changed, it could report the attempted exploitation of a buffer overflow. This of course only detects overflows which try to exploit the original problem that [37, 44] and we address: overwriting of the management information. If an overflow overwrites a pointer in another chunk and no heap operations are called, then the overflow will go undetected.

A major advantage of this approach over [44] is that it does not rely on a global secret value, but can use a per-chunk secret value. While this approach would improve detection of possible attacks, it does not constitute the underlying security principle, meaning that the security does not rely on keeping values in memory secret.

Finally, our countermeasure (as well as other existing ones [23, 44]) focuses on protecting this memory management information, it does not provide strong protection to pointers stored by the program itself in the heap. There are no efficient mechanisms yet to transparently protect these pointers from modification through all possible kinds of heap-based buffer overflows. In order to achieve reasonable performance, countermeasure designers have focused on protecting the most targeted pointers. Extending the protection to more pointers without incurring a substantial performance penalty remains a challenging topic for future research.

6 Related work

Many countermeasures for code injection attacks exist. In this section, we briefly describe the different approaches that could be applicable to protecting against heap-based buffer overflows, but will focus more on the countermeasures which are designed specifically to protect memory allocators from heap-based buffer overflows.

6.1 Protection from attacks on heap-based vulnerabilities

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.

6.1.1 Debugging memory allocators

Dlmalloc has a debugging mode that 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 [42] is a debugging library that will detect both underflows and overflows on heap-allocated memory. It operates by placing each chunk in a separate page and by either placing the chunk at the top of the page and placing a guard page before the chunk (underflow) or by placing the chunk at the end of the page and placing a guard page after the chunk (overflow). This is an effective debugging library but it is not realistic to use in a production environment because of the large amount of memory it uses (every chunk is at least as large as a page, which is 4kb on IA32) and because of the large performance overhead associated with creating a guard page for every chunk. To detect dangling pointer references, it can be set to never release memory back to the system. Instead, Electric fence will mark it as inaccessible, this will however result in an even higher memory overhead.

6.1.2 Runtime allocators

Robertson et al. [44] 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 an information 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 does not stop it from writing into memory that lies above the heap, such as the stack). It also offers a slightly modified version of the Robertson countermeasure as a compile-time option.

ContraPolice [37] 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 [20], 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. While this does protect the contents of other chunks from being overwritten using one of these functions, it provides no protection for other buffer overflows. It also does not protect a buffer from overwriting a pointer stored in the same chunk. This countermeasure can also be bypassed if the canary value can be read: the attacker could write past the canary and make sure to replace the canary with the same value it held before.

Although no performance measurements were done by the author, it is reasonable to assume that the performance overhead would be fairly low.

Recent versions of glibc [23] have added an extra sanity check to its allocator: before removing a chunk from the doubly linked list of free chunks, the allocator checks if the backward pointer of the chunk that the unlinking chunk's forward pointer points to is equal to the unlinking chunk. The same is done for the forward pointer of the chunk's backward pointer. It also adds extra sanity checks that make it harder for an attacker to use the previously described technique of attacking the memory allocator. However, recently, several attacks on this countermeasure were published [43]. Although no data is available on the performance impact of adding these lightweight checks, it is reasonable to assume that no performance loss is incurred by performing them.

DieHard [10] in standalone mode is a memory allocator that will add protection against accidental overflows of buffers by randomizing allocations. The allocator will separate memory management information from the data in the heap. It will also try to protect the contents of a chunk by allocating chunks of a specific chunk size into a region at random positions in the region. This will make it harder for an application to accidentally overwrite the contents of a chunk, however a determined attacker could still exploit it by replicating the modified contents over the entire region. Performance for this countermeasure is very good for some programs (it improves performance for some) while relatively high for others.

6.2 Alternative approaches

Other approaches that protect against the more general problem of buffer overflows also protect against heap-based buffer overflows. In this section, we give a brief overview of this work. A more extensive survey can be found in [58].

6.2.1 Safe languages

Safe languages are languages where it is generally not possible for any known code injection vulnerability to exist as the language constructs prevent them from occurring. A number of safe languages are available that will prevent these kinds of implementation vulnerabilities entirely. Examples of such languages include Java and ML but these are not in the scope of our discussion. However there are safe languages [21, 25, 28, 35, 38, 41] that remain as close to C or C++ as possible, these are generally referred to as safe dialects of C. While some safe languages [18] try to stay more compatible with existing C programs, use of these languages may not always be practical for existing applications.

6.2.2 Compiler-based countermeasures

Bounds checking [5, 30, 45, 54] is the ideal solution for buffer overflows, however performing bounds checking in C can have a severe impact on performance or may cause existing object code to become incompatible with bounds checked object code.

Protection of all pointers as provided by PointGuard [19] is an efficient implementation of a countermeasure that will encrypt (using XOR) all pointers stored in memory with a randomly generated key and decrypts the pointer before loading it into a register. To protect the key, it is stored in a register upon generation and is never stored in memory. However attackers could guess the decryption key if they were able to view several different encrypted pointers. Another attack, described in [3] describes how an attacker could bypass PointGuard by overwriting a particular byte of the pointer. By modifying one byte, the pointer value has changed but the three remaining bytes will still decrypt correctly because of the weakness of XOR encryption. This significantly reduces the randomness (if only one byte needs to be overwritten, an attacker has a 1 in 256 chance of guessing the correct one, if two bytes are overwritten the chances are 1 in 65536, which is still significantly less than 1 in 2^{32}.

Another countermeasure that protects all pointers is the Security Enforcement Tool [55] where runtime protection is performed by keeping a status bit for every byte in memory, that determines if writing to a specific memory region via an unsafe pointer is allowed or not.

6.2.3 Operating system-based countermeasures

Non-executable memory [47, 51] tries to prevent code injection attacks by ensuring that the operating system does not allow execution of code that is not stored in the text segment of the program. This type of countermeasure can however be bypassed by a return-into-libc attack [53] where an attacker executes existing code (possibly with different parameters).

Randomized instruction sets [8, 33] also try to prevent an attacker from executing injected code by encrypting instructions on a per process basis while they are in memory and decrypting them when they are needed for execution. However, software based implementations of this countermeasure incur large performance costs, while a hardware implementation is not immediately practical. Determined attackers may also be able to guess the encryption key and, as such, be able to inject code [49].

Address randomization [13, 51] is a technique that attempts to provide security by modifying the locations of objects in memory for different runs of a program, however the randomization is limited in 32-bit systems (usually to 16 bits for the heap) and as a result may be inadequate for a determined attacker [46].

6.2.4 Library-based countermeasures

LibsafePlus [6] protects programs from all types of buffer overflows that occur when using unsafe C library functions (e..g *strcpy*). It extracts the sizes of the buffers from the debugging information of a program and as such does not require a recompile of the program if the symbols are available. If the symbols are not available, it will fall back to less accurate bounds checking as provided by the original Libsafe [7] (but extended beyond the stack). The performance of the countermeasure ranges from acceptable for most benchmarks provided to very high for one specific program used in the benchmarks

6.2.5 Execution monitoring

In this section we describe two countermeasures that will monitor the execution of a program and will prevent transferring control-flow which could be unsafe.

Program shepherding [34] is a technique that will monitor the execution of a program and will disallow control-flow transfers[11] that are not considered safe. An example of a use for shepherding is to enforce return instructions to only return to the instruction after the call site. The proposed implementation of this countermeasure is done using a runtime binary interpreter, as a result the performance impact of this countermeasure is significant for some programs, but acceptable for others.

Control-flow integrity [1] determines a program's control flow graph beforehand and ensures that the program adheres to it. It does this by assigning a unique ID to each possible control flow destination of a control flow transfer. Before transferring control flow to such a destination, the ID of the destination is compared to the expected ID, and if they are equal, the program proceeds as normal. Performance overhead may be acceptable for some applications, but may be prohibitive for others.

7 Conclusion

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.

Afterwards, we presented a redesign for existing memory allocators that is more resilient to these attacks than existing allocator implementations. We implemented this design by modifying an existing memory allocator. This implementation has been made publicly available. We demonstrated that it has a negligible, sometimes even beneficial, impact on performance. The overhead in terms of memory usage is very acceptable. Although our approach is straightforward, surprisingly, it offers stronger security than comparable countermeasures with similar performance overhead because it does not rely on the secrecy of random numbers stored in memory.

[11]Such a control flow transfer occurs when e.g. a *call* or *ret* instruction is executed.

References

[1] Martin Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, pages 340–353, Alexandria, Virginia, U.S.A., November 2005. ACM.

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

[3] Steven Alexander. Defeating compiler-level buffer overflow protection. *;login: The USENIX Magazine*, 30(3), June 2005.

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

[5] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. Efficient detection of all pointer and array access errors. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 290–301, Orlando, Florida, U.S.A., June 1994. ACM.

[6] Kumar Avijit, Prateek Gupta, and Deepak Gupta. Tied, libsafeplus: Tools for runtime buffer overflow protection. In *Proc. of the 13th USENIX Security Symp.*, San Diego, CA, August 2004.

[7] Arash Baratloo, Navjot Singh, and Timothy Tsai. Transparent run-time defense against stack smashing attacks. In *USENIX 2000 Annual Technical Conference Proceedings*, pages 251–262, San Diego, California, U.S.A., June 2000. USENIX Association.

[8] Elena Gabriela Barrantes, David H. Ackley, Stephanie Forrest, Trek S. Palmer, Darko Stefanović, and Dino Dai Zovi. Randomized instruction set emulation to disrupt binary code injection attacks. In *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS2003)*, pages 281–289, Washington, District of Columbia, U.S.A., October 2003. ACM.

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

[10] Emery D. Berger and Benjamin G. Zorn. Diehard: Probabilistic memory safety for unsafe languages. In *ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation (PLDI 2006)*, Ottawa, Canada, June 2006.

[11] Emery D. Berger, Benjamin G. Zorn, and Kathryn S. McKinley. Composing high-performance memory allocators. In *Proceedings of the ACM SIGPLAN 2001Conference on Programming Language Design and Implementation (PLDI)*, pages 114–124, Snowbird, Utah, U.S.A., June 2001.

[12] Emery D. Berger, Benjamin G. Zorn, and Kathryn S. McKinley. Reconsidering custom memory allocation. In *Proceedings of the 2002 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 1–12, Seattle, Washington, U.S.A., November 2002. ACM, ACM Press.

[13] Sandeep Bhatkar, Daniel C. DuVarney, and R. Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *Proceedings of the 12th USENIX Security Symposium*, pages 105–120, Washington, District of Columbia, U.S.A., August 2003. USENIX Association.

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

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

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

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

[18] Jeremy Condit, Matthew Harren, Scott McPeak, George C. Necula, and Westley Weimer. CCured in the real world. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 232–244, San Diego, California, U.S.A., 2003. ACM.

[19] Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. PointGuard: protecting pointers from buffer overflow vulnerabilities. In *Proceedings of the 12th USENIX Security Symposium*, pages 91–104, Washington, District of Columbia, U.S.A., August 2003. USENIX Association.

[20] 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.

[21] Dinakar Dhurjati, Sumant Kowshik, Vikram Adve, and Chris Lattner. Memory safety without runtime checks or garbage collection. In *Proceedings of the 2003 ACM SIGPLAN Conference on Language, Compiler, and Tool Support for Embedded Systems*, pages 69–80, San Diego, California, U.S.A., June 2003. ACM.

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

[23] Free Software Foundation. The gnu c library. `http://http://www.gnu.org/software/libc`.

[24] Wolfram Gloger. ptmalloc. `http://www.malloc.de/en/`.

[25] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in cyclone. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 282–293, Berlin, Germany, June 2002.

[26] 'Dirk Grunwald, Benjamin Zorn, and Robert Henderson. Improving the cache locality of memory allocation. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation (PLDI)*, pages 177–186, New York, New York, U.S.A., June 1993.

[27] John L. Henning. Spec cpu2000: Measuring cpu performance in the new millennium. *Computer*, 33(7):28–35, July 2000.

[28] Trevor Jim, Greg Morrisett, Dan Grossman, Michael Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of C. In *USENIX Annual Technical Conference*, pages 275–288, Monterey, California, U.S.A., June 2002. USENIX Association.

[29] 'Mark S. Johnstone and Paul R. Wilson. The memory fragmentation problem: Solved? In *Proceedings of the 1st ACM SIGPLAN International Symposium on Memory Management*, pages 26–36, Vancouver, British Columbia, Canada, October 1998. ACM, ACM Press.

[30] Richard W. M. Jones and Paul H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Proceedings of the 3rd International Workshop on Automatic Debugging*, number 009-02 in Linköping Electronic Articles in Computer and Information Science, pages 13–26, Linköping, Sweden, 1997. Linköping University Electronic Press.

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

[32] Michel MaXX Kaempf. Sudo ¡ 1.6.3p7-2 exploit. `http://packetstormsecurity.org/0211-exploits/hudo.c`.

[33] Gaurav S. Kc, Angelos D. Keromytis, and Vassilis Prevelakis. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS2003)*, pages 272–280, Washington, District of Columbia, U.S.A., October 2003. ACM.

[34] Vladimir Kiriansky, Derek Bruening, and Saman Amarasinghe. Secure execution via program shepherding. In *Proceedings of the 11th USENIX Security Symposium*, San Francisco, California, U.S.A., August 2002. USENIX Association.

[35] Sumant Kowshik, Dinakar Dhurjati, and Vikram Adve. Ensuring code safety without runtime checks for real-time control systems. In *Proceedings of the International Conference on Compilers Architecture and Synthesis for Embedded Systems*, pages 288–297, Grenoble, France, October 2002.

[36] 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.

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

[38] James R. Larus, Thomas Ball, Manuvir Das, Robert DeLine, Manuel Fähndrich, Jon Pincus, Sriram K. Rajamani, and Ramanathan Venkatapathy. Righting software. *IEEE Software*, 21(3):92–100, May/June 2004.

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

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

[41] George Necula, Scott McPeak, and Westley Weimer. CCured: Type-safe retrofitting of legacy code. In *Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 128–139, Portland, Oregon, U.S.A., January 2002. ACM.

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

[43] Phantsmal Phantasmagoria. The malloc maleficarum. `http://lists.grok.org.uk/pipermail/full-disclosure/2005-October/037905.html`.

[44] 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.

[45] Olatunji Ruwase and Monica S. Lam. A practical dynamic buffer overflow detector. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium*, San Diego, California, U.S.A., February 2004. Internet Society.

[46] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the Effectiveness of Address-Space Randomization. In *Proceedings of the 11th ACM conference on Computer and communications security*, pages 298–307, Washington, District of Columbia, U.S.A., October 2004. ACM, ACM Press.

[47] Solar Designer. Non-executable stack patch. `http://www.openwall.com`.

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

[49] Nora Sovarel, David Evans, and Nathanael Paul. Where's the FEEB? the effectiveness of instruction set randomization. In *Proceedings of the 14th USENIX Security Symposium*, Baltimore, Maryland, U.S.A., August 2005. Usenix.

[50] W. Richard Stevens. *Advanced Programming in the UNIX enironment*. Addison-Wesley, 1993.

[51] The PaX Team. Documentation for the PaX project. `http://pageexec.virtualave.net/docs/`.

[52] Ruud van der Pas. Memory hierarchy in cache-based systems. Technical Report 817-0742-10, Sun Microsystems, Sant a Clara, California, U.S.A., November 2002.

[53] Rafal Wojtczuk. Defeating Solar Designer's Non-executable Stack Patch. `http://www.insecure.org/sploits/non-executable.stack.problems.html`, 1998.

[54] Wei Xu, Daniel C. DuVarney, and R. Sekar. An Efficient and Backwards-Compatible Transformation to Ensure Memory Safety of C Programs. In *Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 117–126, Newport Beach, California, U.S.A., October-November 2004. ACM, ACM Press.

[55] Suan Hsi Yong and Susan Horwitz. Protecting C programs from attacks via invalid pointer dereferences. In *Proceedings of the 9th European software engineering conference held jointly with 10th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 307–316. ACM, ACM Press, September 2003.

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

[57] Yves Younan. Dnmalloc 1.0. http://www.fort-knox.org, 2005.

[58] 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.

[59] Yves Younan, Wouter Joosen, and Frank Piessens. Efficient protection against heap-based buffer overflows without resorting to magic. In *Proceedings of the International Conference on Information and Communication Security (ICICS 2006)*, Raleigh, North Carolina, U.S.A., December 2006.

[60] Yves Younan, Wouter Joosen, Frank Piessens, and Hans Van den Eynden. Security of memory allocators for C and C++. Technical Report CW419, Departement Computerwetenschappen, Katholieke Universiteit Leuven, July 2005.

[61] Zen-parse. Wu-ftpd 2.6.1 exploit. `http://www.derkeiler.com/Mailing-Lists/securityfocus/vuln-dev/2001-12/0160.html`.