Efficient protection against heap-based buffer overflows without resorting to magic[1]

Yves Younan, Wouter Joosen, and Frank Piessens

DistriNet, Dept. of Computer Science, Katholieke Universiteit Leuven
Celestijnenlaan 200A, B-3001 Heverlee, Belgium
{yvesy,wouter,frank}@cs.kuleuven.be

Abstract. Bugs in dynamic memory management, including for instance heap-based buffer overflows and dangling pointers, are an important source of vulnerabilities in C and C++. Overwriting the management information of the memory allocation library is often a source of attack on these vulnerabilities. All existing countermeasures with low performance overhead rely on magic values or canaries. A secret value is placed before a crucial memory location and by monitoring whether the value has changed, overruns can be detected. Hence, if attackers are able to read arbitrary memory locations, they can bypass the countermeasure. In this paper we present an approach that, when applied to a memory allocator, will protect against this attack vector without resorting to magic. 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 technique that these attackers may employ is the code injection attack, where they are able to insert code into the program's address space and can subsequently execute it. Vulnerabilities that could lead to this kind of attack are still a significant portion of the weaknesses found in modern software systems, especially in programs written in C or C++.

A wide range of vulnerabilities exists that allow an attacker to inject code. The most well-known and most exploited vulnerability is the standard stack-based buffer overflow: attackers write past the boundaries of a stack-based buffer and overwrite the return address of a function so that it points to their injected code. When the function subsequently returns, the code injected by the attackers is executed [1].

[1] Draft version of a paper presented at ICICS 2006 (http://discovery.csc.ncsu.edu/ICICS06/, conference version available from Springer Verlag via http://dx.doi.org/10.1007/11935308_27

However, several other vulnerabilities exist that allow an attacker to inject code into an application. Such vulnerabilities can also occur when dealing with dynamically allocated memory, which we describe in more detail in Section 2. Since no return addresses are available in the heap, an attacker must overwrite other data stored in the heap to inject code. The attacker could overwrite a pointer located in this memory, but since these are not always available in heap-allocated memory, attackers often overwrite the management information that the memory allocator stores with heap-allocated data.

Many countermeasures have been devised that try to prevent code injection attacks [2]. Several approaches try and solve the vulnerabilities entirely [3,4,5,6]. These approaches generally suffer from a substantial performance impact. Others with better performance results have mostly focused on stack-based buffer overflows [7,8,9,10,11].

Countermeasures that protect against attacks on dynamically allocated memory can be divided into four categories. The first category tries to protect the management information from being overwritten by using magic values that must remain secret [12,13] . While these are efficient, they can be bypassed if an attacker is able to read or guess the value based on other information the program may leak. Such a leak may occur, for example, if the program has a 'buffer over-read' or a format string vulnerability. A second category focuses on protecting all heap-allocated data by placing guard pages[2] around them [14]. this however results in chunk size which are multiples of page sizes (which is 4 kb on IA32), which results in a large waste of memory and a severe performance loss (because a separate guard page must be allocated every time memory is allocated). A third category protects against code injection attacks by performing sanity checks to ensure that the management information does not contains impossible values[15]. The fourth category separates the memory management information from the data stored in these chunks. In this paper we propose an efficient approach which falls in the fourth category. It does not rely on magic values and can be applied to existing memory allocators.

To illustrate that this separation is practical we have implemented a prototype (which we call dnmalloc), that is publicly available [16]. Measurements of both performance and memory usage overhead show that this separation can be done at a very modest cost. This is surprising: although the approach is straightforward, the cost compared to existing approaches in the first category is comparable or better while security is improved.

Besides increased security, our approach also implies other advantages: because the often-needed memory management information is stored separately, the pages that only hold the program's data, can be swapped out by the operating system as long as the program does not need to access that data [17]. A similar benefit is that, when a program requests memory, our countermeasure will ensure that it has requested enough memory from the operating system to service the request, without writing to this memory. As such, the operating

---

[2] A guard page is page of memory where no permission to read or to write has been set. Any access to such a page will cause the program to terminate.

system will defer physical memory allocation until the program actually uses it, rather than allocating immediately (if the operating system uses lazy or optimistic memory allocation for the heap [18]).

The paper is structured as follows: Section 2 describes the vulnerabilities and how these can be used by an attacker to gain control of the execution flow using a memory allocator's memory management information. Section 3 describes the main design principles of our countermeasure, while Section 4 details our prototype implementation. In Section 5 we evaluate our countermeasure in multiple areas: its performance impact, its memory overhead and its resilience against existing attacks. In Section 6 we describe related work and compare our approach to other countermeasures that focus on protecting the heap. Section 7 contains our conclusion.

2 Heap-based vulnerabilities

Exploitation of a buffer overflow on the heap is similar to exploiting a stack-based overflow, except that no return addresses are stored in this segment of memory. Therefore, an attacker must use other techniques to gain control of the execution-flow. An attacker could overwrite a function pointer or perform an indirect pointer overwrite [19] on pointers stored in these memory regions, but these are not always available. Overwriting the memory management information that is generally associated with dynamically allocated memory [20,21,22], is a more general way of exploiting a heap-based overflow.

Memory allocators allocate memory in chunks. These chunks typically contain memory management information (referred to as *chunkinfo*) alongside the actual data (*chunkdata*). Many different allocators can be attacked by overwriting the *chunkinfo*. We will describe how dynamic memory allocators can be attacked by focusing on a specific implementation of a dynamic memory allocator called *dlmalloc* [23] which we feel is representative. *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. In this section we will briefly describe some important aspects of *dlmalloc* to illustrate how it can be attacked. We will then demonstrate how the application can be manipulated by attackers into overwriting arbitrary memory locations by overwriting the allocator's *chunkinfo* using two different heap-based programming vulnerabilities.

2.1 Doug Lea's memory allocator

The *dlmalloc* library is a runtime memory allocator that divides the heap memory at its disposal into contiguous chunks. These chunks vary in size as the various allocation routines (*malloc*, *free*, *realloc*, ...) are called. An important property of this allocator is that, after one of these routines completes, a free

chunk never borders on another free chunk, as free adjacent chunks are 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 that size is removed from the list and made available for use in the program (i.e. it turns into an allocated chunk).

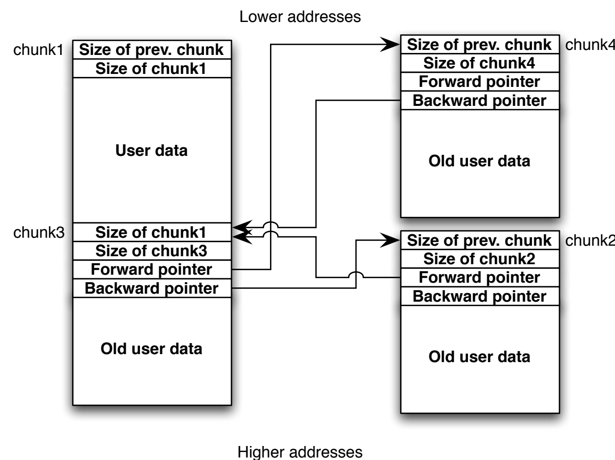

Fig. 1. Heap containing used and free chunks

All memory management information (including this list of free chunks) is stored in-band. That is, the information is stored in the chunks: when a chunk is freed, the memory normally allocated for data is used to store a forward and backward pointer. Figure 1 illustrates what a typical heap of used and unused chunks looks like. *Chunk1* is an allocated chunk containing information about the size of the chunk stored before it and its own size[3]. 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 an arbitrary location 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

[3] 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 last chunk in our example: its forward pointer points to a next chunk in the list and its backward pointer points to *chunk3*.

2.2 Attacks on dynamic memory allocators

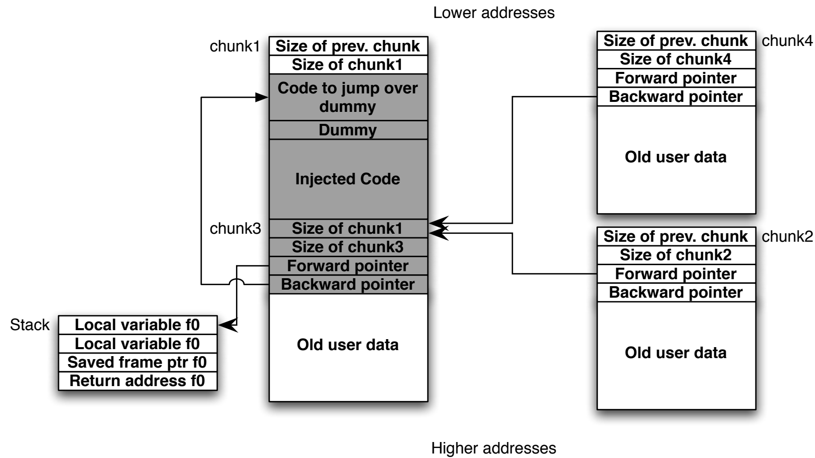

Fig. 2. Heap-based buffer overflow

Figure 2 shows what could happen if an array that is located in *chunk1* is overflowed: an attacker overwrites the management information of *chunk3*. The size fields are left unchanged (although these can be modified if an attacker desires). The forward pointer is changed to point to 12 bytes before function f0's return address, and the backward pointer is changed to point to code that will jump over the next few bytes and then execute the injected code. When *chunk1* is subsequently freed, it is coalesced together with *chunk3* into a larger chunk. As *chunk3* is no longer a separate chunk after the coalescing, it must first be removed from the list of free chunks (this is called unlinking). Internally a free chunk is represented by a datastructure containing the fields depicted in *chunk3* in Fig. 2. A chunk is unlinked as follows:

```
chunk3->fd->bk = chunk3->bk
chunk3->bk->fd = chunk3->fd
```

As a result, the value of the memory location that is twelve bytes (because of the location of the field in the structure) 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 Fig. 2 the return address will 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. This technique can be used to overwrite arbitrary memory locations [20,21].

A similar attack can occur when memory is deallocated twice. This is called a double free vulnerability [25].

3 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 a separate contiguous memory regions that only contains 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 3 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 [26]. However, when larger chunks are requested, it can also allocate memory in the shared memory area [4] using the *mmap*[5] systemcall to allocate memory for the chunk. In Fig. 3, 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. 3 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 Section 3.1 we describe the structures that are used to retrieve the *chunkinfo* when presented with a pointer to *chunkdata*. In Section 3.2, we discuss the management of the region where these *chunkinfos* are stored.

3.1 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

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

[5] 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) [26].

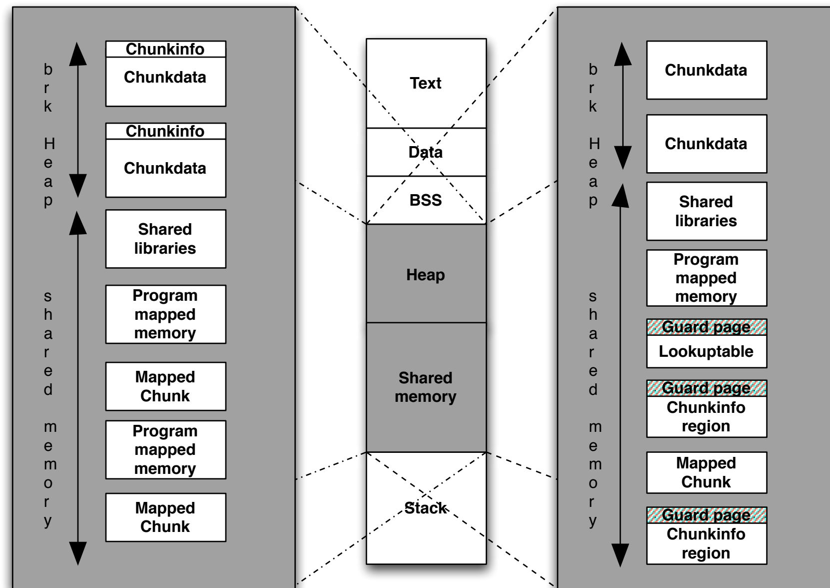

Fig. 3. Original (left) and modified (right) process memory layout

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. 3 we illustrate what the layout looks like in a typical program that uses this design.

3.2 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. 3, 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 can not 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. 3). 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.

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 [27]. 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 *chunkinfo region*.

4 Prototype Implementation

Our allocator was implemented by modifying *dlmalloc 2.7.2* to incorporate the changes described in Section 3. 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.

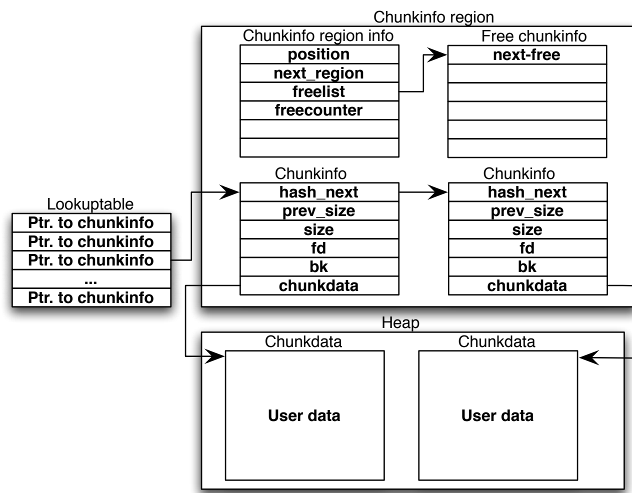

Fig. 4. *Lookup table* and chunkinfo layout

4.1 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 chunksize), 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* which 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. 4).

4.2 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. 4: 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. 4, 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 [28] 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 [29,30,31,32]. 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*.

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

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*[6] 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*.

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

Dnmalloc and all files needed to reproduce these benchmarks are available publicly [16].

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).

SPEC CPU2000 Integer benchmark programs						
Program	DL r/t	DN r/t	R/t overh.	DL mem	DN mem	Mem. overh.
164.gzip	253 ± 0	253 ± 0	0%	180.37	180.37	0%
175.vpr	361 ± 0.15	361.2 ± 0.14	0.05%	20.07	20.82	3.7%
176.gcc	153.9 ± 0.05	154.1 ± 0.04	0.13%	81.02	81.14	0.16%
181.mcf	287.3 ± 0.07	290.1 ± 0.07	1%	94.92	94.92	0%
186.crafty	253 ± 0	252.9 ± 0.03	-0.06%	0.84	0.84	0.12%
197.parser	347 ± 0.01	347 ± 0.01	0%	30.08	30.08	0%
252.eon	770.3 ± 0.17	782.6 ± 0.1	1.6%	0.33	0.34	4.23%
253.perlbmk	243.2 ± 0.04	255 ± 0.01	4.86%	53.80	63.37	17.8%
254.gap	184.1 ± 0.03	184 ± 0	-0.04%	192.07	192.07	0%
255.vortex	250.2 ± 0.04	223.6 ± 0.05	-10.61%	60.17	63.65	5.78%
256.bzip2	361.7 ± 0.05	363 ± 0.01	0.35%	184.92	184.92	0%
300.twolf	522.9 ± 0.44	511.9 ± 0.55	-2.11%	3.22	5.96	84.93%
Allocator-intensive benchmarks						
Program	DL r/t	DN r/t	R/t overh.	DL mem	DN mem	Mem. overh.
boxed-sim	230.6 ± 0.08	232.2 ± 0.12	0.73%	0.78	1.16	49.31%
cfrac	552.9 ± 0.05	587.9 ± 0.01	6.34%	2.14	3.41	59.13%
espresso	60 ± 0.02	60.3 ± 0.01	0.52%	5.11	5.88	15.1%
lindsay	239.1 ± 0.02	242.3 ± 0.02	1.33%	1.52	1.57	2.86%

Table 2. Average macrobenchmark runtime and memory usage results for *dlmalloc* and *dnmalloc*

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.

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 [29]. When running the same tests on a similar system with L1 and L2 cache[7] disabled, the performance benefit for *vortex* went down from 10% to 4.5%.

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*

Microbenchmarks We have included two microbenchmarks. In the first microbenchmark, the time that the program takes to perform 100,000 *mallocs* of random[8] 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*[9] 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.

[7] These are caches that are faster than the actual memory in a computer and are used to reduce the cost of accessing general memory [33].
[8] Although a fixed seed was set so two runs of the program return the same results
[9] This call will fill a particular range in memory with a particular byte.

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.

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 2. They contain the complete overhead of all extra memory the countermeasure uses compared to *dlmalloc*.

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.

Exploit for	Dlmalloc	Dnmalloc
Wu-ftpd 2.6.1 [34]	Shell	Continues
Sudo 1.6.1 [35]	Shell	Crash
Sample heap-based buffer overflow	Shell	Continues
Sample double free	Shell	Continues

Table 4. Results of exploits against vulnerable programs

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 4 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.

These kinds of security evaluations can only prove that a particular attack works, but it can not 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. This will protect from those attacks where the memory management information is used to perform a code injection attack.

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 [12,13] 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 on a chunk, the numbers would be compared and if changed, it could report the attempted exploitation of a buffer overflow. A major advantage of this approach over [12] 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 [15,12]) 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

Countermeasures that protect against attacks on dynamically allocated memory can be divided into three categories. The first category tries to protect the management information from being overwritten by using magic values that must remain secret. While these are efficient, they can be bypassed if an attacker is able to read or guess the value based on other information the program may leak. Such a leak may occur, for example, if the program has a 'buffer over-read' or a format string vulnerability. A second category focuses on protecting all heap-allocated data by placing guard pages around them. this however results in chunk size which are multiples of page sizes, which results in a large waste of memory and a severe performance loss. A third category protects against code injection attacks by performing sanity checks to ensure that the management information does not contains impossible values.

Robertson et al. [12] designed a countermeasure that attempts to protect against attacks on the *ptmalloc* 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 read the encryption key (or the management information together with the checksum). 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.

This countermeasure is efficient, although other benchmarks were used to test the performance overhead in [12], they report similar overhead to ours.

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 [13] 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 [7], 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 [15] 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 which 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 [36]. 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.

Electric fence [14] 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.

6.2 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 [2]

Compiler-based countermeasures Bounds checking [3,4,5,6] 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 [37] 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 [38] describes how an attacker could bypass PointGuard by partially overwriting a pointer. By only needing a partial overwrite, the randomness can be reduced, making a brute force attack feasible (1 byte: 1 in 256, 2 bytes: 1 in 65536, instead of 1 in 2^{32}).

Operating system-based countermeasures Non-executable memory [27,39] 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 [40] where an attacker executes existing code (possibly with different parameters).

Address randomization [27,41] 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 [42].

Library-based countermeasures LibsafePlus [43] 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 [9] (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.

Execution monitoring Program shepherding [44] is a technique that will monitor the execution of a program and will disallow control-flow transfers[10] 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.

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

Control-flow integrity [45] 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 presented a design for existing memory allocators that is more resilient to attacks that exploit heap-based vulnerabilities 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 magic values.

References

1. Aleph One: Smashing the stack for fun and profit. Phrack **49** (1996)
2. Younan, Y., Joosen, W., Piessens, F.: Code injection in C and C++ : A survey of vulnerabilities and countermeasures. Technical Report CW386, Departement Computerwetenschappen, Katholieke Universiteit Leuven (2004)
3. Austin, T.M., Breach, S.E., Sohi, G.S.: Efficient detection of all pointer and array access errors. In: Proc. of the ACM '94 Conf. on Programming Language Design and Implementation, Orlando, FL (1994)
4. Jones, R.W.M., Kelly, P.H.J.: Backwards-compatible bounds checking for arrays and pointers in C programs. In: Proc. of the 3rd Int. Workshop on Automatic Debugging, Linköping, Sweden (1997)
5. Ruwase, O., Lam, M.S.: A practical dynamic buffer overflow detector. In: Proc. of the 11th Network and Distributed System Security Symp., San Diego, CA (2004)
6. Xu, W., DuVarney, D.C., Sekar, R.: An Efficient and Backwards-Compatible Transformation to Ensure Memory Safety of C Programs. In: Proc. of the 12th ACM Int. Symp. on Foundations of Software Engineering, Newport Beach, CA (2004)
7. Cowan, C., Pu, C., Maier, D., Hinton, H., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., Zhang, Q.: StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In: Proc. of the 7th USENIX Security Symp., San Antonio, TX (1998)
8. Etoh, H., Yoda, K.: Protecting from stack-smashing attacks. Technical report, IBM Research Divison, Tokyo Research Laboratory (2000)
9. Baratloo, A., Singh, N., Tsai, T.: Transparent run-time defense against stack smashing attacks. In: USENIX 2000 Technical Conf. Proc., San Diego, CA (2000)
10. Xu, J., Kalbarczyk, Z., Patel, S., Ravishankar, K.I.: Architecture support for defending against buffer overflow attacks. In: Second Workshop on Evaluating and Architecting System dependabilitY, San Jose, CA (2002)

11. Younan, Y., Pozza, D., Joosen, W., Piessens, F.: Extended protection against stack smashing attacks without performance loss. In: Proc. of the Annual Computer Security Apps. Conf., Miami, FL (2006)
12. Robertson, W., Kruegel, C., Mutz, D., Valeur, F.: Run-time detection of heap-based overflows. In: Proc. of the 17th Large Installation Systems Administrators Conf., San Diego, CA (2003)
13. Krennmair, A.: ContraPolice: a libc extension for protecting applications from heap-smashing attacks. http://www.synflood.at/contrapolice/ (2003)
14. Perens, B.: Electric fence 2.0.5. http://perens.com/FreeSoftware/ (1999)
15. Free Software Foundation: GNU C library. http://www.gnu.org/software/libc (2004)
16. Younan, Y.: Dnmalloc 1.0. http://www.fort-knox.org (2005)
17. Kamp, P.H.: Malloc(3) revisted. In: Proc. of the USENIX 1998 Anual technical conference, New Orleans, LA (1998)
18. Summit, S.: Re: One of your c.l.c faq question. Comp.lang.C newsgroup (2001)
19. Bulba, Kil3r: Bypassing Stackguard and stackshield. Phrack **56** (2000)
20. anonymous: Once upon a free(). Phrack **57** (2001)
21. Kaempf, M.: Vudo - an object superstitiously believed to embody magical powers. Phrack **57** (2001)
22. Solar Designer: JPEG COM marker processing vulnerability in netscape browsers. http://www.openwall.com/advisories/OW-002-netscape-jpeg.txt (2000)
23. Lea, D., Gloger, W.: malloc-2.7.2.c. Comments in source code (2001)
24. Gloger, W.: ptmalloc. http://www.malloc.de/en/ (1999)
25. Dobrovitski, I.: Exploit for CVS double free() for linux pserver. http://seclists.org/lists/bugtraq/2003/Feb/0042.html (2003)
26. Stevens, W.R.: Advanced Programming in the UNIX env. Addison-Wesley (1993)
27. The PaX Team: Documentation for PaX. http://pax.grsecurity.net (2000)
28. Henning, J.L.: Spec cpu2000: Measuring cpu performance in the new millennium. Computer **33**(7) (2000)
29. Grunwald, D., Zorn, B., Henderson, R.: Improving the cache locality of memory allocation. In: Proc. of the ACM 1993 Conf. on Programming Language Design and Implementation, New York, NY (1993)
30. Johnstone, M.S., Wilson, P.R.: The memory fragmentation problem: Solved? In: Proc. of the 1st ACM Int. Symp. on Memory Management, Vancouver, BC (1998)
31. Berger, E.D., Zorn, B.G., McKinley, K.S.: Composing high-performance memory allocators. In: Proc. of the ACM Conf. on Programming Language Design and Implementation, Snowbird, UT (2001)
32. Berger, E.D., Zorn, B.G., McKinley, K.S.: Reconsidering custom memory allocation. In: Proc. of the ACM Conf. on Object-Oriented Programming Systems, Languages and Apps., Seattle, WA (2002)
33. van der Pas, R.: Memory hierarchy in cache-based systems. Technical Report 817-0742-10, Sun Microsystems, Sant a Clara, CA (2002)
34. Zen-parse: Wu-ftpd 2.6.1 exploit. Vuln-dev mailinglist (2001)
35. Kaempf, M.: Sudo exploit. Bugtraq mailinglist (2001)
36. Phantasmagoria, P.: The malloc maleficarum. Bugtraq mailinglist (2005)
37. Cowan, C., Beattie, S., Johansen, J., Wagle, P.: PointGuard: protecting pointers from buffer overflow vulnerabilities. In: Proc. of the 12th USENIX Security Symp., Washington, DC (2003)
38. Alexander, S.: Defeating compiler-level buffer overflow protection. ;login: The USENIX Magazine **30**(3) (2005)

39. Solar Designer: Non-executable stack patch. http://www.openwall.com (1998)
40. Wojtczuk, R.: Defeating Solar Designer's Non-executable Stack Patch. Bugtraq mailinglist (1998)
41. Bhatkar, S., DuVarney, D.C., Sekar, R.: Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In: Proc. of the 12th USENIX Security Symp., Washington, DC (2003)
42. Shacham, H., Page, M., Pfaff, B., Goh, E.J., Modadugu, N., Boneh, D.: On the Effectiveness of Address-Space Randomization. In: Proc. of the 11th ACM Conf. on Computer and communications security, Washington, DC (2004)
43. Avijit, K., Gupta, P., Gupta, D.: Tied, libsafeplus: Tools for runtime buffer overflow protection. In: Proc. of the 13th USENIX Security Symp., San Diego, CA (2004)
44. Kiriansky, V., Bruening, D., Amarasinghe, S.: Secure execution via program shepherding. In: Proc. of the 11th USENIX Security Symp., San Francisco, CA (2002)
45. Abadi, M., Budiu, M., Erlingsson, U., Ligatti, J.: Control-flow integrity. In: Proc. of the 12th ACM Conf. on Computer and Communications Security, Alexandria, VA (2005)