Applying machinemodel-aided countermeasure design to improve memory allocator security

Yves Younan, Wouter Joosen, Frank Piessens
DistriNet, Department of Computer Science
Katholieke Universiteit Leuven
Celestijnenlaan 200a, B-3001 Leuven, Belgium
{yvesy,wouter,frank}@cs.kuleuven.ac.be

Abstract

This paper is a companion paper for the talk that will be presented at the 22nd Chaos Communication Congress. We will describe the background on how the results that we detail in the talk were achieved but will not substantially overlap with the talk. We will focus on a more structured approach to build countermeasures using a model of the execution environment. This machinemodel allows reasoning about countermeasures at a higher level and allows for a more effective design where possible shortcomings can be spotted more easily. The paper then describes how we applied this technique to design a countermeasure to protect memory allocators from heap-based attacks.

1 Introduction

Code injection attacks have been a known security problem for over 20 years, yet they still occur in modern day applications and countermeasures that try to protect against these attacks are often built ad hoc and as a result are often bypassable by attackers that use more advanced exploitation techniques. In this paper we will discuss a more structured approach to designing countermeasures for code injection attacks that was first described in [9]. While stack-based buffer overflows have dominated the vulnerabilities which can cause code injection attacks, heap-based buffer overflows and dangling pointer references to heap memory are also important avenues of attack. In this paper we describe how we applied our approach to build a countermeasure for attacks on heap-based vulnerabilities that take advantage of properties of the memory allocator.

The paper is structured as follows: section 2 briefly summarizes the technical details of how an attacker would exploit a heap-based vulnerability when the application uses dlmalloc as memory allocator. Section 3 describes our model-based approach to designing countermeasures for code injection attacks and how we applied this approach to build a more secure allocator. Section 4 presents our conclusion.

2 Heap-based vulnerabilities

Heap memory is dynamically allocated at run-time by the application. Exploitation of a buffer overflow in this memory is similar to exploiting a stack-based overflow, except that no return addresses are stored in this segment of memory so an attacker must use other techniques to gain control of the execution-flow. An attacker could of course overwrite a function pointer or perform an indirect pointer overwrite [3] on pointers stored in these memory regions, but these are not always available. Overwriting the memory management information that is generally associated with a dynamically allocated chunk that is managed by a dynamic memory allocator [1, 2, 5, 7] is a more general way of exploiting a heap-based overflow.

We will demonstrate how dynamic memory allocators can be attacked by focusing on a specific implementation of a dynamic memory allocator called *dlmalloc* [6]. While dlmalloc is used as a basis for the allocator in the GNU/Linux operating system, these techniques could also be applied to similar allocators used in other operating systems (as we demonstrate in [10]). We will describe *dlmalloc* briefly and will summarize two attack techniques that would allow an attacker to manipulate the application into overwriting arbitrary memory locations by overwriting the allocator's memory management information.

2.1 Doug Lea's memory allocator

The *dlmalloc* library is a run-time memory allocator that divides the heap memory at its disposal into contiguous chunks, 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 that size in the list will be removed from the list and will be made available for use in the program (i.e. it will turn into an allocated chunk).

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 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[1]. The rest of the chunk is available for the program

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

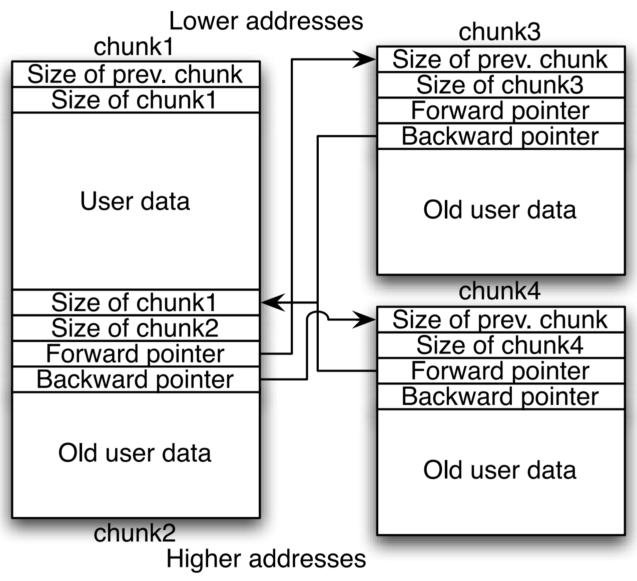

Figure 1: Heap containing used and free chunks

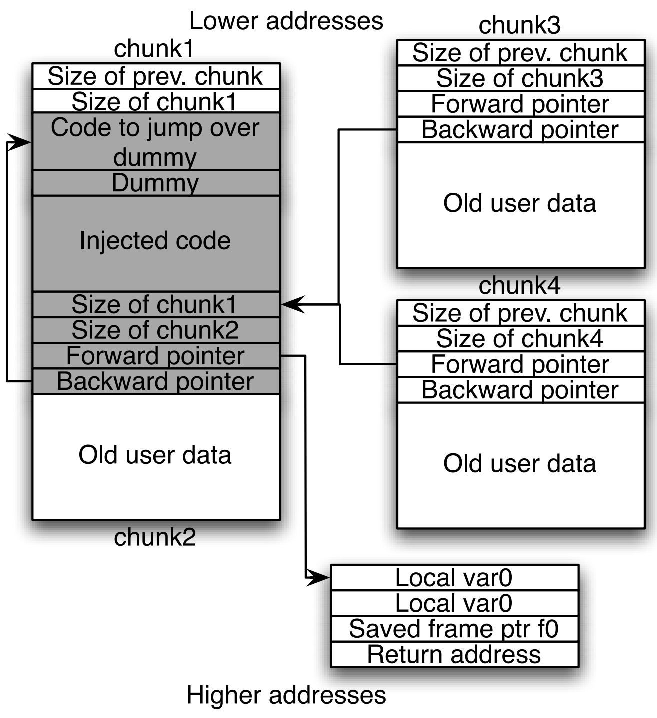

Figure 2: Heap-based buffer overflow

to write data in. *Chunk2* [2] represents a free chunk that is located in a doubly linked list together with *chunk3* and *chunk4*. *Chunk3* is the first chunk in the chain: its backward pointer points to *chunk2* and its forward pointer points to a previous chunk in the list. *Chunk2* is the next chunk, with its forward pointer pointing to *chunk3* and its backward pointer pointing to *chunk4*. *Chunk4* is the last chunk in our example: its backward pointer points to a next chunk in the list and its forward pointer points to *chunk2*.

2.2 Exploiting heap-based overflows

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

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

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

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

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

2.3 Exploiting dangling pointer references

A pointer to a memory location could refer to a memory location that has been deallocated either explicitly by the programmer (e.g., by calling free) or by code generated by the compiler (e.g., a function epilogue, where the stackframe of the function is removed from the stack). Dereferencing of this pointer is generally unchecked in a C compiler, causing the dangling pointer reference to become a problem. In normal cases this would cause the program to crash or exhibit uncontrolled behavior as arbitrary locations could be overwritten. However, double free vulnerabilities are a specific version of the dangling pointer reference problem that could lead to exploitation. A double free vulnerability occurs when already freed memory is deallocated a second time. This could again allow an attacker to overwrite arbitrary memory locations [4].

this coalescing can be abused is explained later.

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

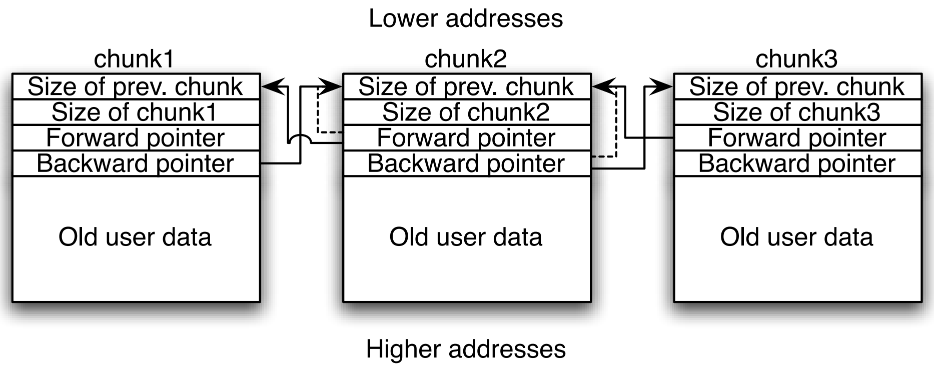

Figure 3: List of free chunks: full lines show a normal list of chunks, dotted lines show the changes after a double free has occurred.

We illustrate this using dlmalloc in Figure 3. The full lines in this figure are an example of what the list of free chunks of memory might look like when using the *dlmalloc* memory allocator. *Chunk1* is bigger 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 the following would happen (substitutions made on the code listed above):

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

The forward and backward pointers of *chunk2* both point to itself. The dotted lines in Figure 3 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 3) of the chunk for its own use.

Attackers can now use the same technique that we previously discussed to exploit the heap-based overflow (see Figure 2): they set the forward pointer to point 12 bytes before the return address and change the value of the backward pointer to point to code that will jump over the bytes that will be overwritten. When the program tries to allocate a chunk of the same size again (or tries to free this one), it will again try to unlink *chunk2* which will overwrite the return address with the value of *chunk2's* backward pointer.

3 Model-based countermeasure design

In the previous section we described how heap-based buffer overflows and dangling pointer references could be exploited when using dlmalloc. Similar techniques can be used to exploit these type of vulnerabilities on other allocators. In [10] we analyzed 5 different memory allocators and showed how these vulnerabilities could be exploited.

Most countermeasures that have been proposed (see [8] for a detailed discussion) use an ad hoc approach when trying to prevent vulnerabilities. In [9] we described a more methodological approach to countermeasure design by building a model of the execution environment of a system based on the memory locations and abstractions that could influence the execution flow. The model contains all the addresses and abstractions that could be modified by an attacker to directly or indirectly influence the execution flow of a program. This information is supplemented with locations that could lead to indirect pointer overwriting and contextual information. The contextual information describes what the information contained in the model is used for at a particular place in the execution flow and what operations are performed on it. Such a model for a particular platform is called a machinemodel and allows a countermeasure designer to build countermeasures at a more abstract level.

By applying this method to dlmalloc, a countermeasure was designed which protects the memory management information associated with heap-allocated memory from misuse by using code injection attacks.

3.1 Countermeasure design for heap-based attacks

The datastructures and abstractions that are contained in the machinemodel are represented by a UML-diagram. Specific datastructures are represented as classes, its datamembers represent the datastructures stored in this datastructure and member functions denote which operations can be performed on this memory. The datamembers contain extra information in the form of specific signs to denote the order and frequency that particular members can occur in within this structure.

+ denotes that the datamembers in this class are ordered.

- denotes that the order of datamembers in the class does not matter.

* denotes that the part of memory can occur zero or more times, other datamembers occur exactly once.

We will first describe a machinemodel for Doug Lea's malloc and afterwards this machinemodel will be modified to build our countermeasure called DistriNet malloc.

3.1.1 Machinemodel for dlmalloc

Figure 4 contains a machinemodel for the heap when using dlmalloc. The heap contains zero or more malloc chunks and the order of these chunks differs from program to program (denoted by the -* before the malloc chunk). At this level, one operation can be performed: allocate memory for a chunk. Mallocchunk represents a chunk, it contains a prevsize and a size and the allocator has 2 views on chunks, depending on if they are allocated or not. As such at the top level, 2 operations can be performed: free and reallocate. Allocated chunks contain user data and in normal circumstances only a free call should be called on it. Free chunks contain a forward and backward pointer which build up a linked list of free chunks. A reallocate operation can be called on a free chunk so it can be reused by the program or it can be coalesced into a larger chunk.

3.1.2 Modified machinemodel

On most architectures, code and data are stored in separate parts of memory and have different properties. By applying the same principles to separate pointers and control flow information from normal data we can protect these from modification by code injection attacks. When applied to the heap-based memory allocator we can separate the management information from the normal data. This will make the allocator more resilient against the earlier described attacks. Figure 5 shows the new memory layout while Figure 6 is a modified machinemodel of the dlmalloc-model that describes the way the countermeasure works.

For the technical and performance details of the implementation of this modified allocator we refer the reader to the presentation or [10].

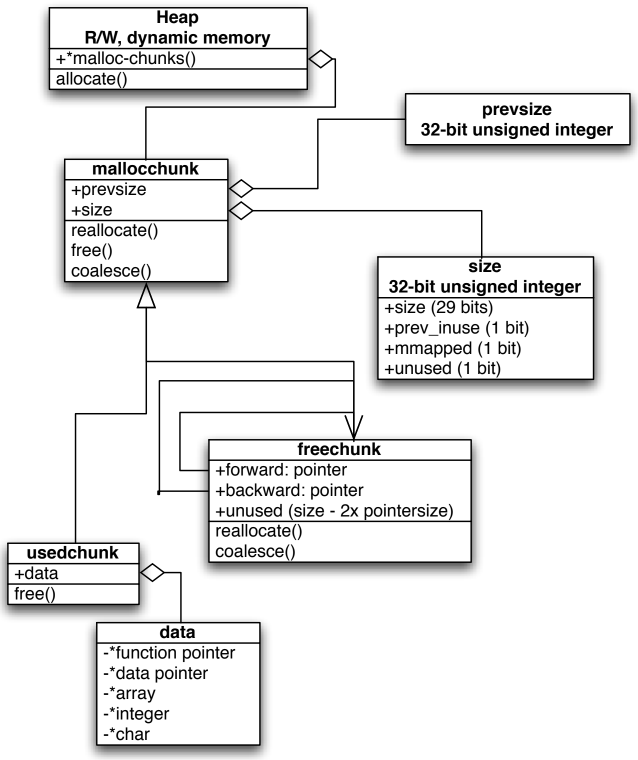

Figure 4: Machinemodel for the heap when using dlmalloc

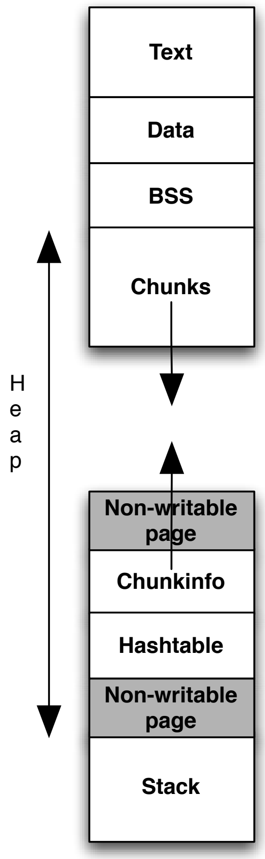

Figure 5: New memory layout

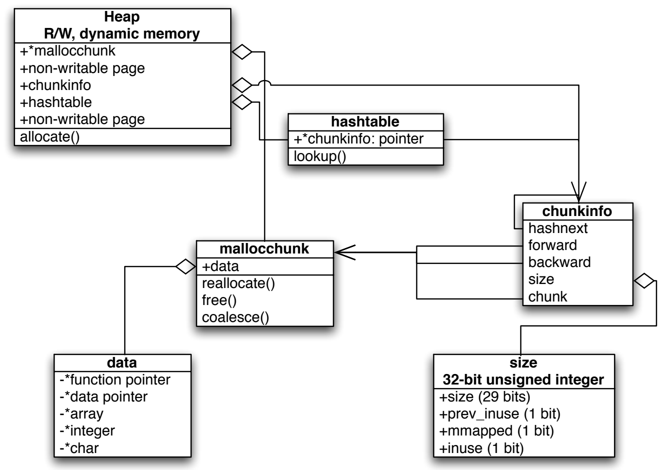

Figure 6: Machinemodel for the countermeasure

4 Future Work and Conclusion

An important limitation in our approach, that in general also applies to related countermeasures, is that it does not protect pointers stored on the heap that do not belong to the allocator. This would still allow attackers to either overwrite a function pointer stored on the heap or allow them to perform an indirect pointer overwrite on a data pointer. This is an important limitation and we will address this by also separating pointer information from the rest of the data.

This separation can be accomplished by mapping a memory area where only pointers will be stored. Pointers for a particular chunk will be stored sequentially in this area. To access these pointers, two extra fields would be added to the chunk information: a ptrcount field (to denote the number of pointers in the chunk) and a ptrptr which points to the chunk's first pointer in the memory area. These modifications to the memory allocator require modifications to the compiler so that the correct pointer is accessed when the program makes use of such a pointer. However, such a modification would make the use of the allocator less transparent because it can no longer be dynamically deployed: all programs that use the allocator would need to be recompiled. Moreover, careful analysis is needed to ensure that the countermeasure does not break existing programs.

We also plan to apply model-based countermeasure design to other areas that attackers target for code injection attacks, like the stack and datasegments. These countermeasures would also use the idea of separating normal data from execution flow data. The details of these countermeasures are described in [9].

A next step in the model-based approach is to build a metamodel which allows the building of machinemodels by system (but not necessarily security) experts. This reduces the initial cost of using the approach to build a countermeasure and allows for better collaboration: the person building the model is not necessarily the person designing the countermeasure. Such a metamodel will also ensure uniformity of machinemodels which can be useful when porting countermeasures between platforms.

The use of machinemodels allows countermeasaure designers to build countermeasures in a more structured manner. The higher level of abstraction offered by such a model, allows the designer to focus on the problem while being able to ignore implementation details until implementation time. We demonstrated how to apply this model-based design to build a countermeasure for heap-based code injection attacks. The implementation of our countermeasure has a negligible impact on performance and memory usage while still being effective against attacks. It also does not suffer from some of the shortcomings of other countermeasures (it does not rely on memory secrecy). The implementation is currently fully operational: it can run console-based applications, X, gnome, etc... It will be released during the 22nd Chaos Communication Congress and will be available for download at http://www.fort-knox.org/.

References

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

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

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

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

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

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

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

[8] 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.

[9] Yves Younan, Wouter Joosen, and Frank Piessens. A methodology for designing countermeasures against current and future code injection attacks. In *Proceedings of the Third IEEE International Information Assurance Workshop 2005 (IWIA2005)*, College Park, Maryland, U.S.A., March 2005. IEEE, IEEE Press.

[10] 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.