Future Generation Computer Systems 28 (2012) 1252–1257

Contents lists available at SciVerse ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Securing heap memory by data pointer encoding[☆]

Kyungtae Kim, Changwoo Pyo [*]

Hongik University, 72-1 Sangsudong, Mapoku, Seoul, 121-791, Republic of Korea

ARTICLE INFO

Article history:
Received 31 October 2010
Received in revised form
27 January 2011
Accepted 3 February 2011
Available online 3 March 2011

Keywords:
Data pointer encoding
Heap memory
Heap overflow attack
Dual-linked list

ABSTRACT

Since pointer variables frequently cause programs to crash in unexpected ways, they often pose vulnerability abused as immediate or intermediate targets. Although code pointer attacks have been historically dominant, data pointer attacks are also recognized as realistic threats. This paper presents how to secure heap memory from data pointer attacks, in particular, heap overflow attacks. Our protection scheme encrypts the data pointers used for linking free chunks, and decrypts the pointers only before dereferencing. We also present a list structure with duplicate links that is harder to break than the conventional linked list structure. Our experiment shows that the proposed data pointer encoding is effective and has slightly better performance than the integrity check of link pointers in GNU's standard C library.

1. Introduction

Since pointer variables frequently cause programs to crash in unexpected ways, they often pose vulnerability abused as immediate or intermediate targets. Although code pointer attacks such as stack smashing [1] have been historically dominant, data pointer attacks are also recognized as realistic threats [2].

Well-known vulnerability of data pointers can be found in the heap area managed by dynamic memory allocators. For instance, the bind8 attack [3] on name servers overflows the heap area compromising two pointers used for linking free chunks. When the dynamic memory manager accesses the compromised data pointers for housekeeping, the target memory location whose address is held in one of the data pointers is overwritten with the value in another data pointer. Another well-known vulnerability in heap space may be observed by freeing a chunk twice [4]. Deallocating an already freed chunk corrupts the free chunk list. Attackers can take advantage of the double-free vulnerability to initiate data pointer attacks.

This paper presents how to secure heap memory from data pointer attacks, in particular, heap overflow attacks. In our scheme, the dynamic memory manager encrypts the pointers linking free chunks, and decrypts the pointers only when it is necessary to know the real addresses before dereferencing. We also present a list structure with duplicate links that is harder to break than the conventional linked list structure with a single link. In addition, lists with duplicate links enable intrusion detection to work under explicit control. We implemented our idea in the GNU C library and compared its effectiveness and runtime overhead with those of the GNU's standard version in a Linux environment.

In the rest of this paper, Section 2 explains the vulnerabilities of data pointers in heap space and how to attack them. Protection of data pointers by encoding is presented in Section 3. Section 4 is about dual-linked lists. Section 5 demonstrates the effectiveness and performance of data pointer encoding with our implementation in the GNU C library's dynamic memory manager. Section 6 overviews techniques for data pointer protection and Section 7 concludes this paper.

2. Vulnerability of data pointers in heap space

Many Linux systems adopt Doug Lea's memory allocator [5] as the default heap manager. The heap memory space consists of allocated and free chunks shown in Fig. 1. The `prev_size` and `size` fields denote the size of previous and current chunks respectively. Using them, physically adjacent chunks can be accessed. For allocated chunks, only the size field is valid, while all fields are valid for free chunks. The `P` field stands for the PREV_INUSE flag that indicates whether the previous chunk is allocated or not. If `P` is 0, then the previous chunk is a free one. Otherwise the previous chunk is an allocated one. The `prev_size` field is valid only when `P` is zero. The actual area used by a program begins from below the `size` field and ends at the `prev_size` field of the following chunk. Two data pointers, `fd` (forward pointer) and `bk` (back pointer), connect the next and previous free chunks to form doubly linked lists, called *bins*.

[☆] This work was supported by the IT R&D Program of MKE/KEIT [2010-KI002090, Development of Technology Base for Trustworthy Computing] and 2009 Hongik University Research Grant.

[*] Corresponding author. Tel.: +82 2 320 1691; fax: +82 2 332 1653.
E-mail address: pyo@hongik.ac.kr (C. Pyo).

0167-739X/$ – see front matter © 2011 Elsevier B.V. All rights reserved.
doi:10.1016/j.future.2011.02.006

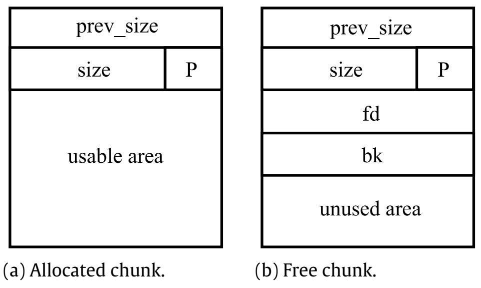

Fig. 1. Structures of memory chunks.

Table 1
Characteristics of bins.

Bin	Structure	Chunk size	Insert/delete
Fast	Singly linked	Uniform	LIFO
Small	Doubly linked	Uniform	FIFO
Large	Doubly linked	Variable (sorted)	Random
Unsorted	Doubly linked	Variable (unsorted)	FIFO

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

Fig. 2. UNLINK macro for separation of a chunk P.

There are four kinds of bins: fast bins, small bins, an unsorted bin and large bins. Fast bins are singly linked lists for fast operation and work as stacks. Each of the fast bins has chunks of the same size. Small bins are used in a first-in-first-out manner. The sizes of the chunks of a small bin are also identical. A large bin keeps free chunks whose sizes range over certain intervals, and the chunks are sorted according to their sizes. When a chunk is allocated from a large bin, the oldest chunk with the least waste is allocated.

The unsorted bin is used as a “cache” for immediate reuse of freed chunks. It imposes no order among its chunks. Memory manager appends a freed chunk at the end of the bin. When memory allocation is requested, the memory allocator searches a fast or small bin with proper chunk size. If there is no chunk available, then the memory allocator resorts to the unsorted bin. If the size of the first chunk can satisfy the request, the chunk is allocated. Otherwise, the chunk is deleted from the unsorted bin and put in another appropriate bin. If the first available chunk is too large, it is split into two chunks. One is allocated, and the other is replaced in a proper bin. The inspection and replacement of cached chunks are repeated until a chunk of proper size is allocated or they are exhausted. Table 1 summarizes the characteristics of each bin.

If a chunk physically adjacent to a free one is released, they are merged together to minimize fragmentation. The merger also simplifies the manipulation of free chunks, but incurs some overhead. The merged chunk would be inserted into the unsorted bin. When deleting a chunk from a bin, the macro UNLINK in Fig. 2 is invoked.

Fig. 3 shows possible phases of deallocation of chunk A by the function `free`. When chunk A is released, chunks A and P should be merged if chunk P next to chunk A is a free one as in Fig. 3(a). To merge chunks A and P, chunk P is deleted from the bin it belongs to. Fig. 3(b) shows the state when chunk P is unlinked from the bin. Fig. 3(c) shows the state when chunks A and P are merged. The merged chunk is kept in the unsorted bin. Dotted links in Fig. 3(d) connect chunks in the unsorted bin.

Fig. 4 shows a code segment causing heap overflow. We assume that chunk b is located right after chunk a. We also assume that the allocated size of chunk a is exactly 104 bytes including the 8-byte header. Note that the size of an allocated chunk may be greater than the requested size depending on the source of the chunk. For

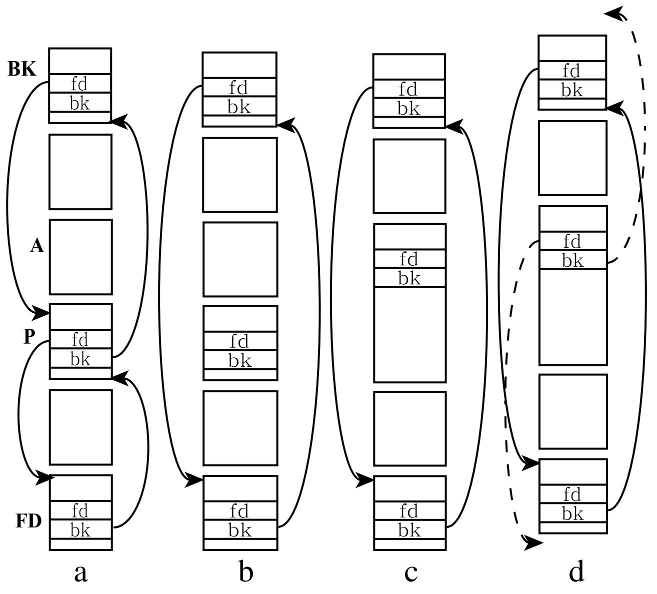

Fig. 3. Phases of deallocation of chunk A by a call to function `free`. Empty squares are allocated chunks. Arrows are the link between two free chunks in a bin.

```
...
a = malloc(96);
b = malloc(80);
c = malloc(80);
strcpy(a, argv[1]);
free(a);
free(b);
free(c);
...
```

Fig. 4. An example of buffer overflow in the heap area.

example, a chunk from the unsorted bin may have space more than requested. Since the function `strcpy` does not limit the size of the second argument, a command line input larger than 112 bytes overwrites the link pointers of chunk b. Fig. 5 shows the state of chunk b when call `strcpy()` is completed. Forward pointer `fd` of chunk b is set to a value less than the address of a return address pointer by 12. Similarly, back pointer `bk` is set to the address of a shellcode.

When the call `free(a)` is executed, flag P of chunk c is checked if chunk b is free. Access to chunk c requires reading the `size` field of chunk b that is compromised to a value so that chunk b is faked as chunk c. Since the P flag of chunk b is 0, function `free` regards chunk b as a free one and chunk b is unlinked to be merged with chunk a. Unlinking chunk b works with the compromised value of the pointers `fd` and `bk` writing the return address slot with the address of a shellcode overwritten by heap overflow. Attacks with a similar pattern can also overwrite an entry of a global offset table for shared objects or other code pointers. The vulnerability of