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.

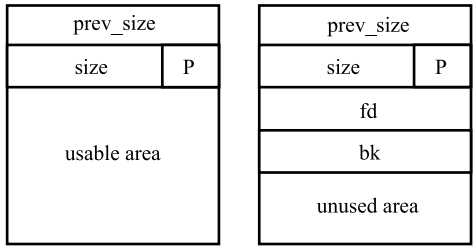
2. Vulnerability of data pointers in heap space

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



(a) Allocated chunk. (b) Free chunk.

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

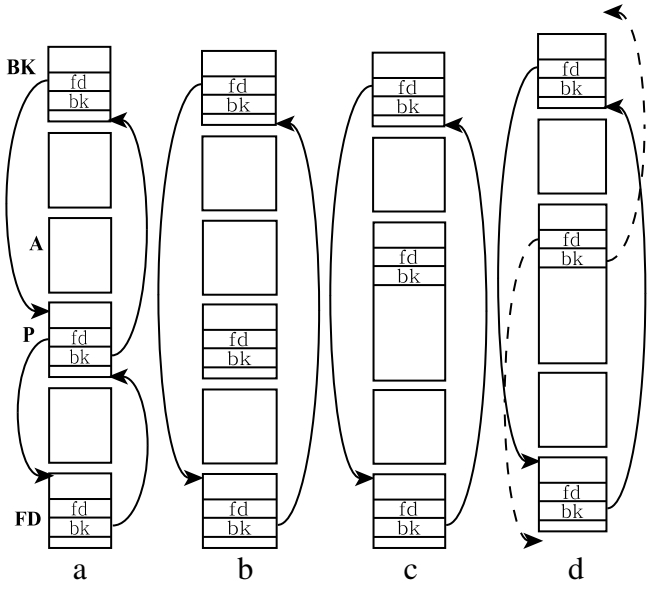


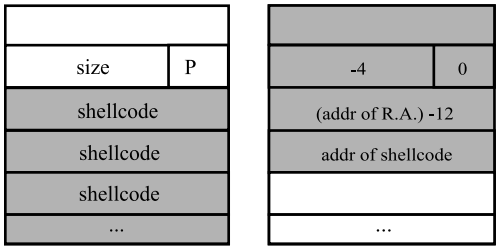
Fig. 3. Phases of deallocation of chunk A by a call to function `free`. Empty squares are allocated chunks. Arrows are the link between two free chunks in a bin.

```
...
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.

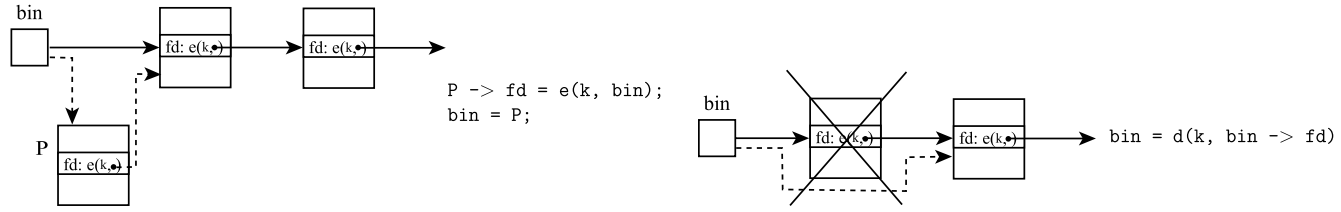
example, a chunk from the unsorted bin may have space more than requested. Since the function `strcpy` does not limit the size of the second argument, a command line input larger than 112 bytes overwrites the link pointers of chunk `b`. Fig. 5 shows the state of chunk `b` when call `strcpy()` is completed. 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



(a) The chunk a. (b) The chunk b.

Fig. 5. Overflowed chunks.



(a) Insertion. (b) Deletion.

Fig. 6. Operations of singly linked lists. e is an encoding function and k is a key.

data pointers makes it possible to compromise code pointers. If the compromised code pointers are accessed, the attack succeeds.

3. Data pointer encoding for linked lists

Bins for free chunks are either singly or doubly linked lists. If a bin is constructed as a singly linked list, only the forward link `fd` is used. For doubly linked lists, both `fd` and `bk` pointers are used. In any case, link pointers can protect themselves by encryption.

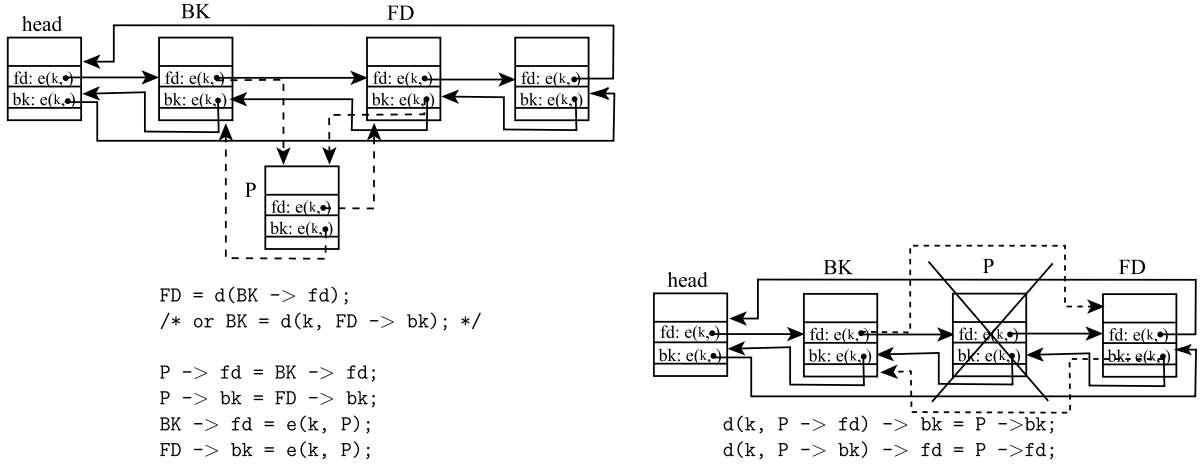
In our scheme, the dynamic memory manager encrypts a pointer linking free chunks immediately after it is *defined*, that is, assigned with an address, and decrypts the pointer only when it is necessary to know the real addresses, before dereferencing. When the link pointer is simply copied to other pointers, it is neither encrypted nor decrypted.

Attackers may overwrite link pointers with values they desire. Unless they encrypt the values used for overwriting pointers with the correct keys, the overwritten pointer values are useless because the decryption would give unpredictable values.

Fig. 6 shows the insertion and deletion operations in a fast bin working as a stack. For insertion, the encoded address of the bin is copied into the `fd` field of chunk P in order to be inserted, and the bin is set to the address of chunk P. Deletion also occurs at the head of the bin. The `fd` field of the first chunk is decrypted and assigned to the bin.

Other bins have the structure of a doubly linked list with a header element, where both the `fd` and the `bk` link fields are encrypted. Fig. 7 shows the insertion and deletion operations. For insertion, chunk P is inserted between the two chunks BK and FD, in which one or both of FD and BK are given. The address of chunk P is encoded by an encoding function `e` and a key `k`. Given BK, the next chunk FD is accessed after decoding it by a decoding function `d` and the key `k`. Similarly, chunk BK can be accessed from chunk FD. Deletion of chunk P simply copies encoded pointer values to the destinations as shown in Fig. 7(b). It needs to decode the pointer values to get the real target addresses.

For encryption and decryption, we use the exclusive-OR (XOR) operation. The decision is made by considering performance as well as security, since too much overhead renders data pointer encoding impractical. Guessing both a and b from e is difficult given that $e = a$ XOR b. With 32-bit keys, the probability of a successful



(a) Insertion. (b) Deletion.

Fig. 7. Operations of doubly linked list with encrypted link pointers.

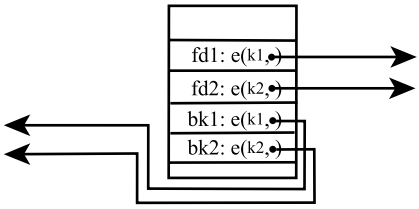


Fig. 8. Memory chunk with dual-link fields.

break is $1/2^{32}$. XOR operations can be done by a single instruction in microprocessors.

Cryptographic keys should be generated from true random number generators. Pseudo-random number generators produce random numbers algorithmically from a seed. For example, cellular automata can generate pseudo-random numbers [6]. If the seed is disclosed, then the subsequent random numbers can be correctly anticipated. Our random key generator uses DRAM as a chaotic source. DRAM should periodically refresh every cell to maintain voltage level. Since DRAM does not allow access during refresh operation, DRAM access initiated during the refresh period would experience a delay of unpredictable length. Randomness can be captured by counting the number of elapsed cycles as DRAM is accessed intensively in a short period. Details can be found in [7]. Generated keys are stored in read-only page to prevent overwriting. Though not perfect, configuring the key storage as "read-only" is an effective and efficient way of protection. Since overwriting the key locations is impossible, attackers should try to read the keys. However, heap attack to read the keys required for successful heap attack is infeasible. Attackers may count on other methods, but the countermeasures against them are beyond the scope of this paper.

4. Encoded dual-linked list

Traditional linked lists have a *mono-link* field for each target. Thus, a singly linked list has a link field for the next element, and a doubly linked list has two link fields for the next and previous elements. With data pointer encoding of mono-linked lists, attacks end up with a segment fault because of memory access violations. To detect and handle actively compromised link pointers, we may use the *dual-link* structure for free chunks shown in Fig. 8.

Free chunks of dual-linked lists use two link fields that possess independently encrypted addresses from the same address. When inserting or deleting a chunk, link fields are decoded and compared with each other. If the decoded values are not the same, it is highly probable that the link pointers are compromised. Fig. 9 and Fig. 10 show code segments for insertion and deletion operations of singly and doubly dual-linked lists.

The dynamic memory managing functions in the GNU's C library check for pointer integrity by using semantic information such as chunk size and flag value. For doubly linked lists, it traces a forward pointer followed by chasing the corresponding back pointer to check that tracing ends at the starting chunk. Although the checking procedure may be efficient and elaborated, it is difficult to apply to data pointers in general and affirm if a checking procedure is sufficient.

Encoding the data in headers may possibly fortify security, but it does not significantly level up protective power whereas execution time overhead increases. It is difficult to succeed in heap attacks without forging the link pointers. When the memory allocator copies the compromised link to another for housekeeping, a critical location is set to the value that attackers want.

5. Experiment

We have modified the dynamic memory allocation functions in the GNU's standard C library version 2.10.1 [8] by replacing the code for checking data pointer integrity with our data pointer encoding. We compared ours with the GNU's with respect to performance and effectiveness in defending against heap overflow attacks. We confirmed that data pointer encoding is as effective as GNU's dynamic memory manager. Also, simulated attacks of slapper worm [9] and attacks using double free [4] could not succeed with our version of a memory allocator.

To test for execution performance, chunks are allocated and deallocated intensively in each bin. We also tested memory allocation requesting chunks whose sizes are greater than the page size 4 Kbytes. The testing load is a mixture of eight times allocation and deallocation respectively. The number of processor cycles is counted in a short period to exclude the possibility of interrupt or other system-wide activity that may shadow the speedups. Table 2 summarizes the overhead with respect to the memory allocation without any validity checking. The means and

```
P -> fd1 = e(k1, bin);
P -> fd2 = e(k2, bin);
bin = P;
```

(a) Insertion.

```
if( d(k1, bin -> fd1) != d(k2, bin -> fd2) )
   exception_handler();
bin = d(k1, bin -> fd1);
```

(b) Deletion.

Fig. 9. Operations on singly linked lists with encoded dual-links.

```
FD = d(k1, BK -> fd1);
if( FD != d(k2, BK -> fd2))
  exception_handler();
/* or BK = d(FD -> bk); */

P -> fd1 = BK -> fd1;
P -> fd2 = BK -> fd2;
if(d(k1, FD ->bk1) != d(k2, FD ->bk2))
  exception_handler();
P -> bk1 = FD -> bk1;
P -> bk2 = FD -> bk2;
BK -> fd1 = e(k1, P);
BK -> fd2 = e(k2, P);
FD -> bk1 = BK -> fd1;
FD -> bk2 = BK -> fd2;
```

(a) Insertion.

```
FD = d(k1, P -> fd1);
if( FD != d(k2, P -> fd2) )
  exception_handler();
FD -> bk1 = P ->bk1;
FD -> bk2 = P ->bk2;

BK = d(k1, P -> bk1);
if( BK != d(k2, P -> bk2) )
  exception_handler();
BK -> fd1 = P ->fd1;
BK -> fd2 = P ->fd2;
```

(b) Deletion.

Fig. 10. Operations on doubly linked lists with encoded dual-links.

Table 2
Overheads of execution cycles for data pointer encoding.

Bin	Mean	Std. dev.	GNU's (%)	Mono (%)	Dual (%)
Fast	162819.06	2641.81	4.63	2.68	3.73
Small	164667.40	2619.87	4.46	1.74	3.12
Large	164480.66	2608.20	4.80	1.20	2.80
LARGE	221288.94	5509.99	3.39	1.31	3.23

Table 3
SPEC CPU2006 integer results. Each column presents execution times and speedups. The GNU's, Mono-link and Dual-link show the execution times and speedups by GNU's standard C library version 2.10 and modified versions using mono-linked and dual-linked lists respectively.

Benchmark	Execution time (s)		
	GNU's	Mono-link (speedup)	Dual-link (speedup)
Perlbench	646.070	643.708 (1.00367)	647.252 (0.99817)
Bzip2	951.453	945.027(1.00680)	951.442 (1.00001)
Gcc	565.421	561.748 (1.00654)	564.281 (1.00202)
Mcf	558.006	556.129 (1.00338)	557.932 (1.00013)
Gobmk	781.067	780.849 (1.00028)	781.575 (0.99935)
Hmmer	1182.578	1182.274 (1.00026)	1175.323 (1.00617)
Sjeng	915.136	894.861 (1.02266)	907.923 (1.00795)
Libquantum	1317.494	1303.694 (1.01059)	1317.360 (1.00010)
H264ref	1097.676	1096.349 (1.00121)	1094.597 (1.00281)
Omnetpp	628.642	622.887 (1.00924)	628.990 (0.99945)
Astar	806.748	810.623 (0.99522)	806.688 (1.00008)
Xalancbmk	587.519	583.609 (1.00670)	588.555 (0.99824)
G-mean		1.00552	1.00120

standard deviations are for the memory allocator without integrity checking. The last three columns show the overheads. Both the mono-encoding and dual-encoding allocators show better performance than GNU's. The last row of Table 2 is a large bin tested with chunks of size greater than the page size 4 Kbytes.

In benchmark testing using SPEC CPU2006, we measured the speedups over GNU C library's memory allocator. Data pointer encoding showed slightly better performance than GNU's. Table 3 shows average speedups of 1.00552 and 1.00120 for mono-link and dual-link bins respectively.

Since the speedups are marginal, it needs to be confirmed if they can survive statistical noise. We collected twelve execution times for each benchmark program and memory allocator. We used four versions of memory allocators. They are basic memory allocator without any checking, glibc's memory allocator, the mono-encoding memory allocator, and the dual-encoding memory allocator. We performed statistical hypothesis testing for the difference of means with significance level 0.05. Table 4 shows if each data set is statistically the same. We also profiled each benchmark to see the portion of execution time for the basic memory allocator. As shown in the second column of Table 4, the portion of execution time for the memory allocator is very small. Nine out of twelve programs spend a negligible portion of execution time for memory allocation. The remaining three spend less than 6% for dynamic memory allocation. It would not be easy to observe speedups when execution lasts long enough for system-wide activity to intervene with a heavy overhead such as input and output. Note that the benchmark `omnetpp` with the largest percentile execution time for memory allocator differentiates a mean from another.

6. Related work

Heap server [10] is an independent process responsible for dynamic memory allocation. Applications request memory allocation through interprocess communication. Heap server stores and manages memory chunk's housekeeping data corresponding to the headers and pointers of Lea's dlmalloc [5] giving the effect of separating the chunk's meta-data from program data. Heap server performs better than dlmalloc in symmetric multiprocessor environment, but it performs much differently for a program after another in benchmarking. Heap server management utilizes randomization in layout and chunk recycling. Its memory overhead is the size of virtual address space, that is, 4 Gbytes for 32-bit processors.

Table 4
Result of statistical testing for the difference of means. The second column is the percentile execution time spent by the memory allocator without pointer checking for each benchmark. *B*, *G*, *M*, and *D* stand for base, GNU's, mono-encoding, and dual-encoding memory allocators. 'Y' means statistical equality.

Benchmark	% time	B=G	B=M	B=D	G=M	G=D	M=D
Perlbench	2.248	N	Y	Y	N	N	Y
Bzip2	0.0	N	N	N	Y	N	N
Gcc	0.332	Y	Y	Y	Y	N	N
Mcf	0.0	Y	Y	Y	Y	Y	Y
Gobmk	0.011	Y	N	Y	N	Y	N
Hmmer	0.025	N	Y	Y	Y	Y	Y
Sjeng	0.0	Y	N	Y	N	Y	N
Libquantum	0.0	Y	N	N	Y	N	Y
H264ref	0.001	Y	Y	Y	Y	Y	Y
Omnetpp	5.425	N	N	N	Y	N	N
Astar	0.092	Y	Y	N	Y	N	N
Xalancbmk	2.296	Y	Y	N	Y	N	N
No		4	5	5	3	7	6
Yes		8	6	6	9	5	3

Chunk's information can be stored separately from program data in the same virtual address space [11]. Non-writable pages protect chunks' management data from being flooded. Adopting the guard page increases space overhead when the memory manager allocates many small chunks. Guard pages can defend against overflow attacks, but are vulnerable to non-linear attacks other than buffer overflow. The above two methods based on the separation of heap management data have the limitation that they produce another module or entity to be protected to protect another. It should be verified that the virtual address space of heap server is secure and that guard pages are sufficient to protect heap management data.

Address space randomization [12] puts randomness in the layout of data objects and gaps between them so that outside attackers cannot guess target locations easily. But its effectiveness is known to be limited [13].

Data space randomization [14] encrypts the values of program variables when they are *defined*, and decrypts the values when they are *used*. It is effective in defense against non-control data attacks. However, its full application to every variable incurs high overheads on performance, and the limitation on accuracy of pointer alias determines its effectiveness. For heap protection, it encrypts application data rather than the link pointers contrary to our approach, and several ad hoc measures are required to wrap up loose ends due to aliasing.

DieHard [15] is developed to minimize memory errors using randomization and replication. It can be used to protect heap from buffer overflow. Memory allocator places a chunk at randomized location in a region according to the chunk's size. A region corresponds to a bin of the memory allocator. More than one duplicates of a region is installed and chunks are allocated or deallocated redundantly in all replicas. The duplicates differently randomize the locations of chunks. Read and write operations on the allocated chunks are performed in parallel. When data is read copies of the values from the duplicates are compared with each other. If they are different, then the chunks are regarded as contaminated. The idea of a replica is similar to ours, but our replication is limited to the forward and back pointers. Our mono- and dual-encoded linked lists do not have space overheads because the link pointers are set up in the area for program data.

There is another approach to non-control data attack that detects abnormal states by behavior monitoring. Behavior monitoring usually traces system calls and additional information such as parameter values. SIDAN [16] proposes a behavior modeling based on the states of non-control data, and instruments programs to detect inconsistency based on the result of static analysis.

7. Conclusion

Data pointer encoding is effective in protecting heap memory. Dual-linked lists make it possible to detect actively compromised link pointers. Performance testing shows that data pointer encoding has less overhead than the heap manager functions of GNU's standard C library.

The idea of data pointer encoding can be applied to self-protection of any pointers whether they are code pointers or data pointers. If a data pointer is involved in program security, the pointer can be protected by data pointer encoding. If a data pointer needs to be protected, a copy of the pointer is installed and the two pointers are encrypted using independent keys. When the pointer is dereferenced, decrypted values of the pointer and its copy are compared to ensure safety. Data pointer encoding can be a simple but powerful programming technique for secure coding.

Acknowledgements

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.

References

[1] A. One, Smashing the stack for fun and profit, Phrack Magazine, 49 (7).
[2] S. Chen, J. Xu, E.C. Sezer, P. Gauriar, R.K. Iyer, Non-control-data attacks are realistic threats, in: the 14th USENIX Security Symposium, 2005, pp. 177–192.
[3] US-CERT, Multiple Vulnerabilities in BIND, 2002. http://www.cert.org/advisories/CA-2002-31.html.
[4] US-CERT, Double Free Bug in zlib Compression Library, 2002. http://www.cert.org/advisories/CA-2002-07.html.
[5] D. Lea, A Memory Allocator, 2009. http://g.oswego.edu/dl/html/malloc.html.
[6] S.-U. Guan, S. Zhang, Pseudorandom number generation based on controllable cellular automata, Future Generation Computer Systems 20 (4) (2004) 627–641.
[7] C. Pyo, S. Pae, G. Lee, DRAM as source of randomness, Electronics Letters 45 (1) (2009) 26–27.
[8] S. Loosemore, R. Stallman, R. McGrath, A. Oram, U. Drepper, The GNU C library reference manual, Free Software Foundation, 2007.
[9] F. Perriot, P. Szor, An analysis of the slapper worm exploit, 2003.http://securityresponse.symantec.com/avcenter/reference/analysis.slapper.worm.pdf.
[10] M. Kharbutli, X. Jiang, Y. Solihin, G. Venkataramani, M. Prvulovic, Comprehensively and efficiently protecting the heap, in: The 12th Architectural Support for Programming Languages and Operating Systems, ACM, 2006, pp. 207–218.
[11] Y. Younan, W. Joosen, F. Piessens, Efficient protection against heap-based buffer overflows without resorting to magic, in: Information and Communications Security, 2006, pp. 379–398.
[12] T.P. Team, Address Space Layout Randomization (ASLR), 2003. http://pax.grsecurity.net/docs/aslr.txt.
[13] H. Shacham, M. Page, B. Pfaff, E. Goh, N. Modadugu, D. Boneh, On the effectiveness of address-space randomization, in: The 11th ACM Conference on Computer and Communications Security, 2004, pp. 298–307.
[14] S. Bhatkar, R. Sekar, Data space randomization, in: Detection of Intrusions and Malware, and Vulnerability Assessment, 2008, pp. 1–22.
[15] E. Berger, B. Zorn, DieHard: probabilistic memory safety for unsafe languages, in: ACM SIGPLAN Conference on Programming Language Design and Implementation, 2006, pp. 158–168.
[16] J. Demay, E. Totel, F. Tronel, SIDAN: A tool dedicated to software instrumentation for detecting attacks on non-control-data, in: Risks and Security of Internet and Systems, 2009, pp. 51–58.

Kyungtae Kim is a student of the Master's program in the Department of Computer Engineering, Hongik University in Seoul, Korea. His research interests include the trustworthiness of programs, tools for program security, virtualization and parallel programming.

Changwoo Pyo received his Ph.D. in Computer Science from the University of Illinois at Urbana-Champaign in 1989. He was a research fellow in the US Army Corps of Engineers in 1990. He has been a professor of the Department of Computer Engineering, Hongik University in Seoul, Korea since 1991. During the period, he served Hongik University as the Chief of Information and Computing Office of Hongik University. He is currently a member of the board of trustees, Korea Institute of Information Scientists and Engineers. His research interests include the trustworthiness of programs, tools for program security, and program analysis and transformation.