KULeuven
DistriNet
Research Group

Memory Allocator Security

Yves Younan, Wouter Joosen, Frank Piessens and Hans Van den Eynden
DistriNet, Department of Computer Science
Katholieke Universiteit Leuven
Belgium
Yves.Younan@cs.kuleuven.ac.be

Overview

- Introduction
- Attacks
- Memory Allocators
- A Safer Allocator
- Related Work
- Conclusion

Introduction

- Many allocators ignore security
- Performance and waste limitation are more important
- Many allocators can be abused to perform code injection attacks
- More security is possible at a modest cost

Overview

- Introduction
- Attacks
  - Heap-based buffer overflow
  - Off by One/Off by Five
  - Dangling Pointer References
- Memory Allocators
- A Safer Allocator
- Related Work
- Conclusion

Heap-based buffer overflows

- Heap memory is dynamically allocated at run-time
- Can also be overflowed but no return address is available
- Modify data pointers (IPO) or function pointers - not always available
- Modify the memory management information associated with heap-allocated memory

Off by one / Off by few bytes

- Special case of buffer overflow: limited space needed
- Off by one: write one byte past the bounds
- Usually only exploitable on little endian machines

(LSB is stored before MSB)

- Off by few bytes: don't occur often but demonstrate low-space attacks

Dangling pointer references

- Pointers to memory that is no longer allocated
- Dereferencing is unchecked in C
- Generally leads to crashes (SIGSEGV)
- Can be used for code injection attacks when deallocated twice (double free)
- A double free can be used to change memory management information allowing an overwrite of arbitrary memory locations

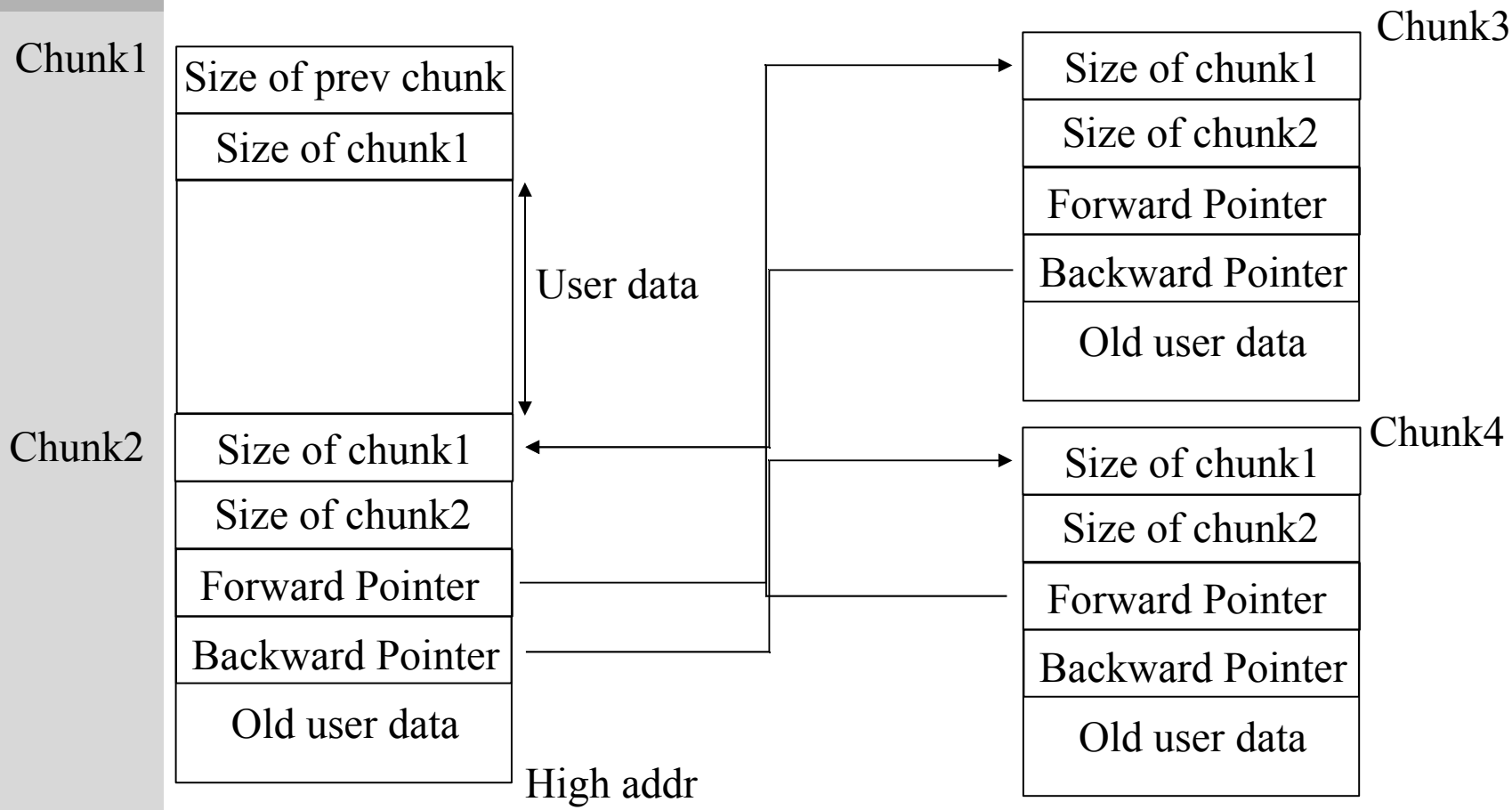
Overview

- Introduction
- Attacks
- Memory Allocators
  - Doug Lea's malloc (Linux)
  - CSRI/Quickfit
  - Poul-Henning Kamp's malloc (BSD)
  - Boehm's garbage collector
- A Safer Allocator
- Related Work
- Conclusion

Doug Lea's malloc

- GNU lib C is based on this malloc
- Every allocation is represented by a chunk
- Management information stored before the chunk
- Free chunks stored in doubly linked list of free chunks
- Two bordering free chunks are coalesced into a larger free chunk
- Description based on dlmalloc 2.7.2
- Attack on it first described by Solar Designer

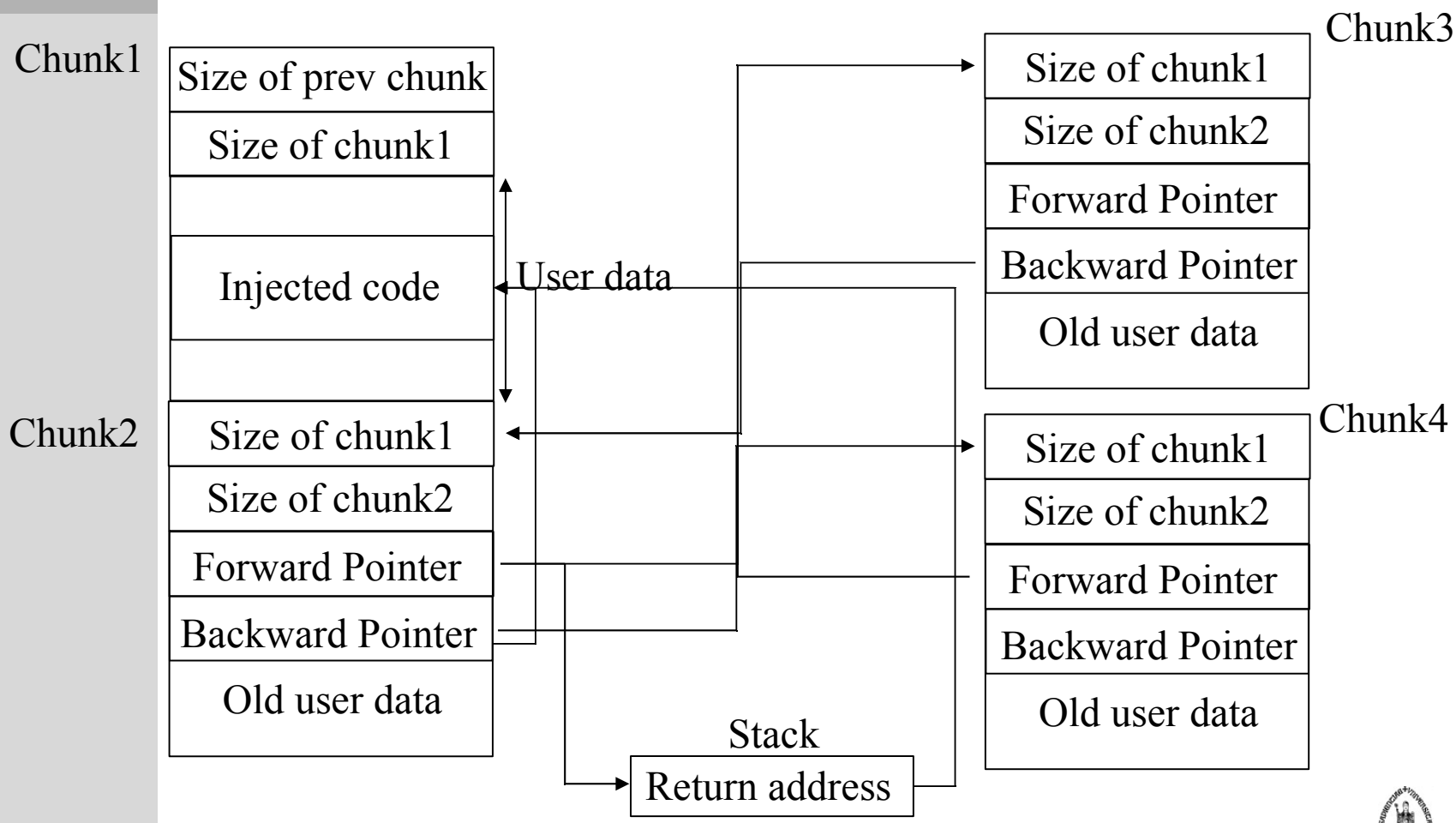
Doug Lea's malloc



Doug Lea's malloc

- Unlink (remove element from the doubly linked list:
  - #define unlink(P, BK, FD) {
    \
    FD = P->fd;
    \
    BK = P->bk;
    \
    FD->bk = BK;
    \
    BK->fd = FD;

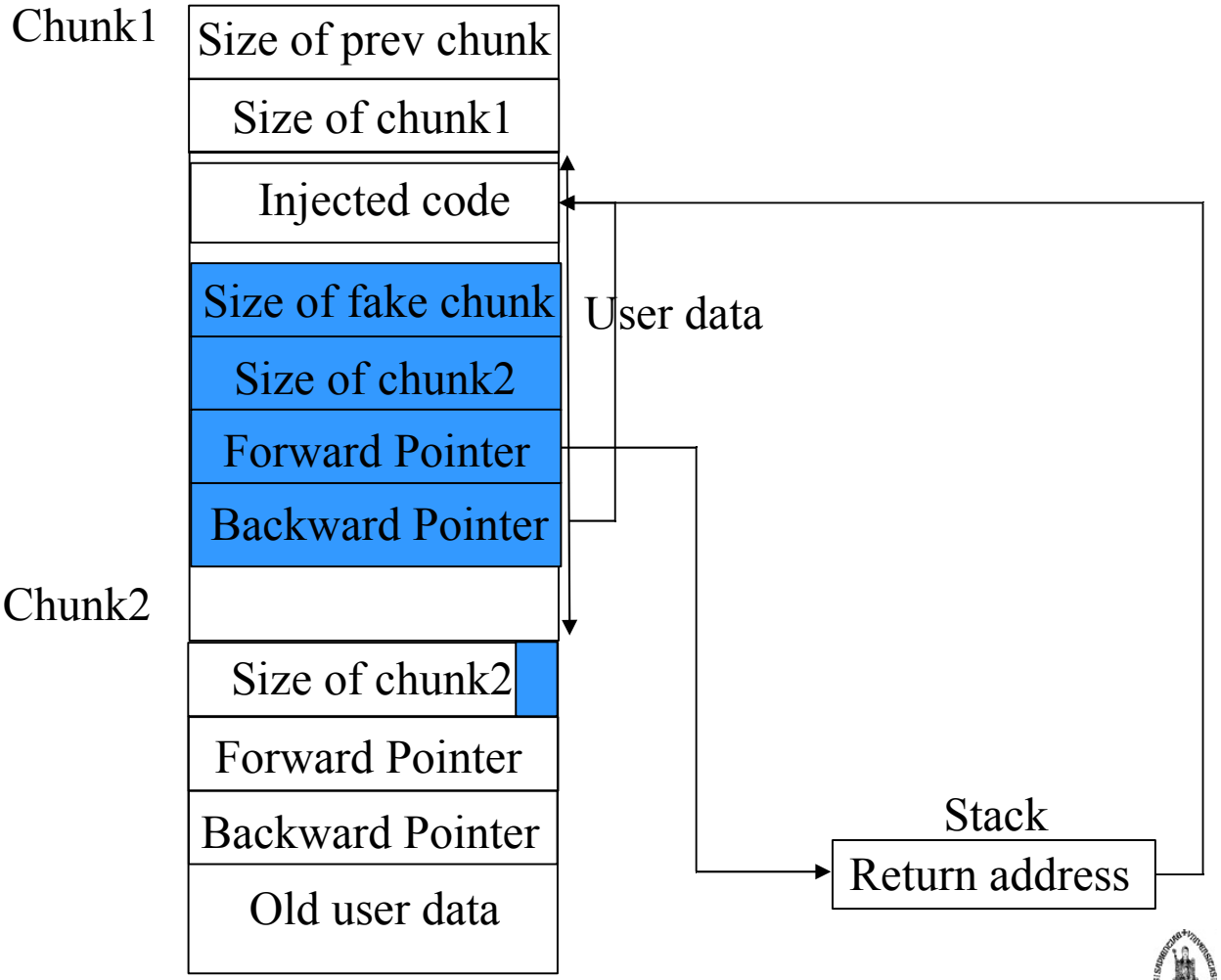
Heap Overflow (dlmalloc)



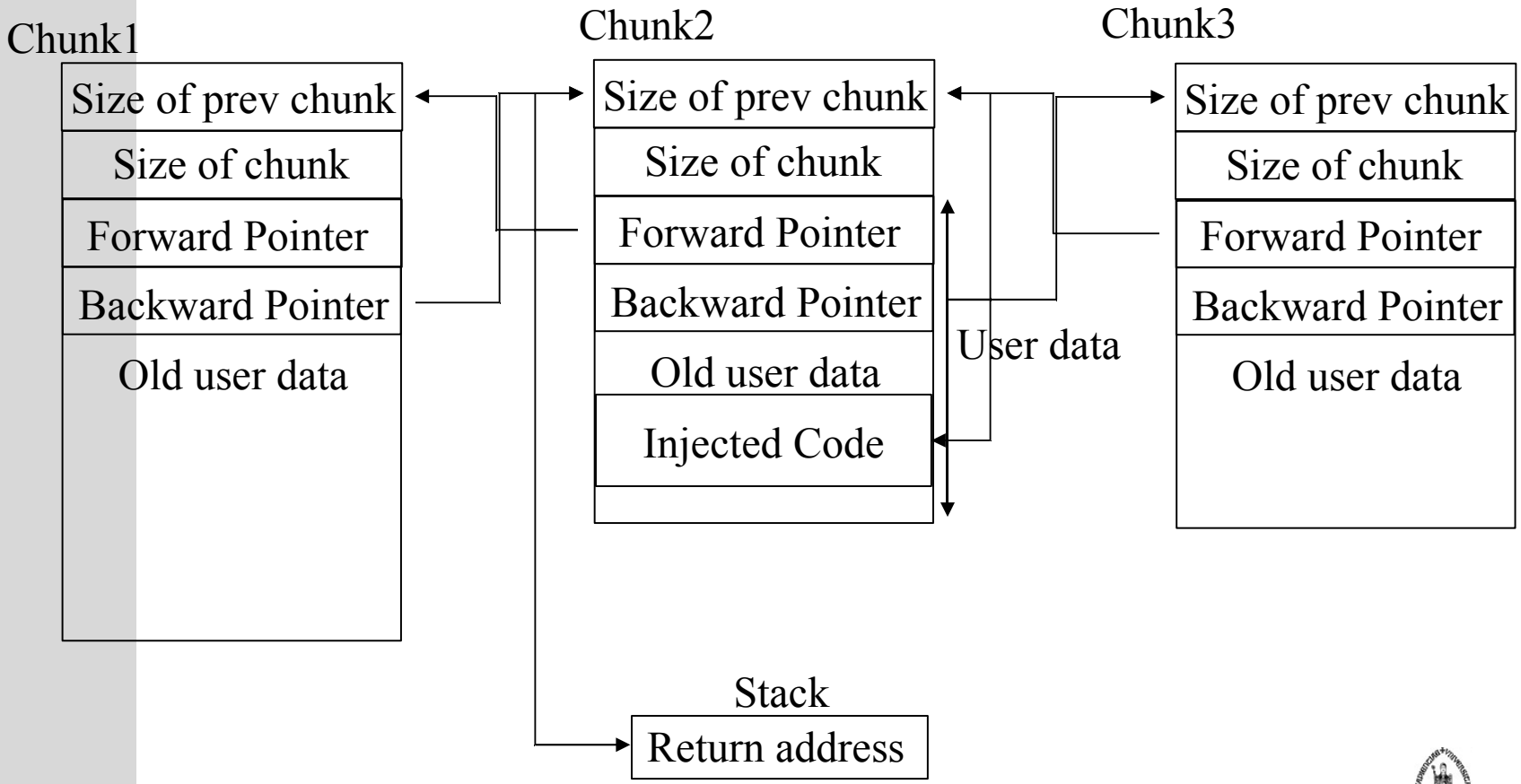
Off by one (dlmalloc)

- Chunk sizes are multiples of 8
- Size contains two flags mmapped and prev_inuse
- Two chunks must be next to each other (no padding) for off by one
- Prev_size of next will be used for data
- Overwrite 1 byte of the size and set prev_inuse to 0 and set a smaller size for prev_size
- Make a fake chunk, containing modified pointers

Off by one (dlmalloc)



Dangling pointer references (dlmalloc)



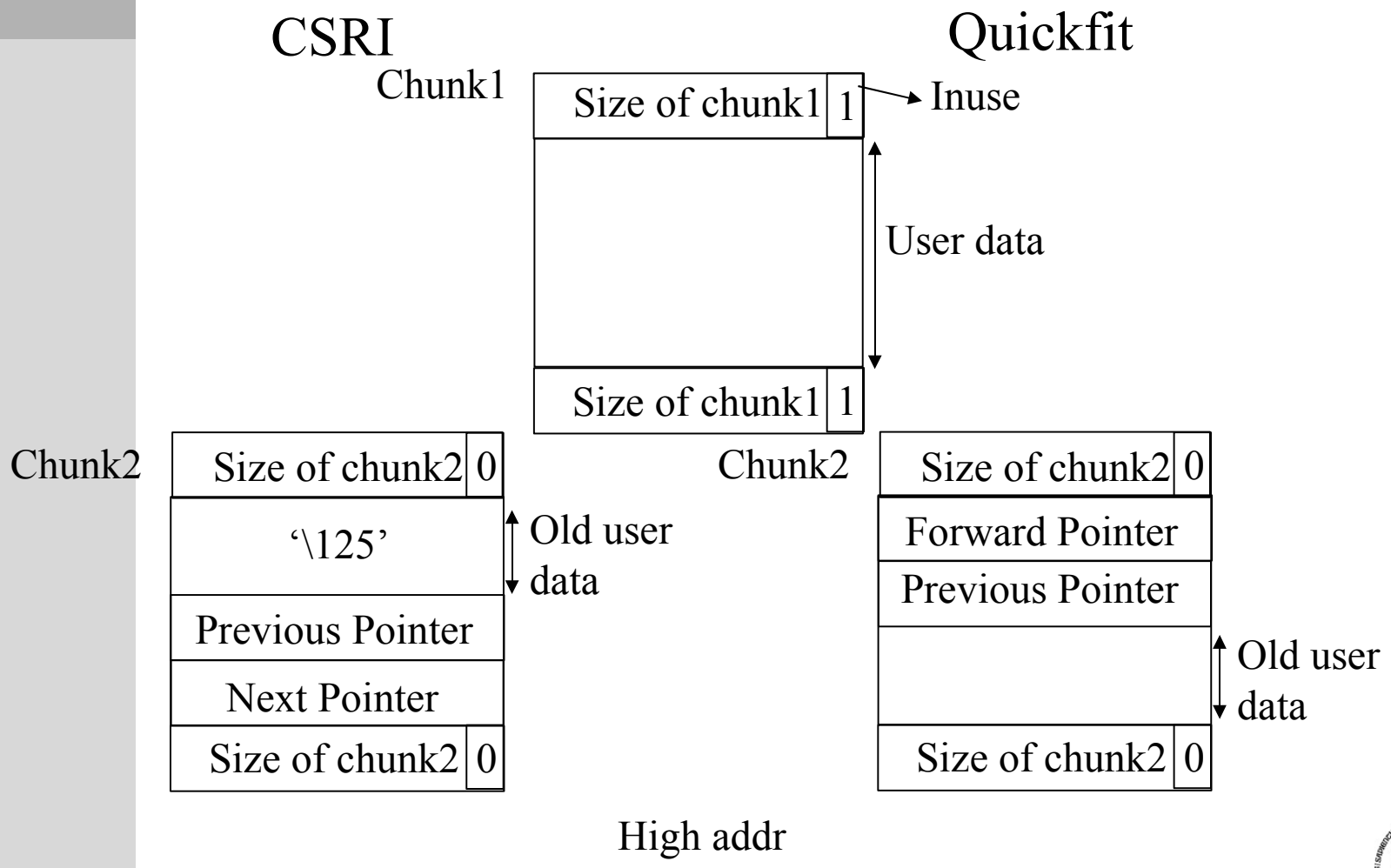
Doug Lea's malloc conclusion

- Vulnerable to:
  - Heap overflow
  - Off by one/five
  - Double free
- Version 2.8.x contains some mitigation techniques, see related work

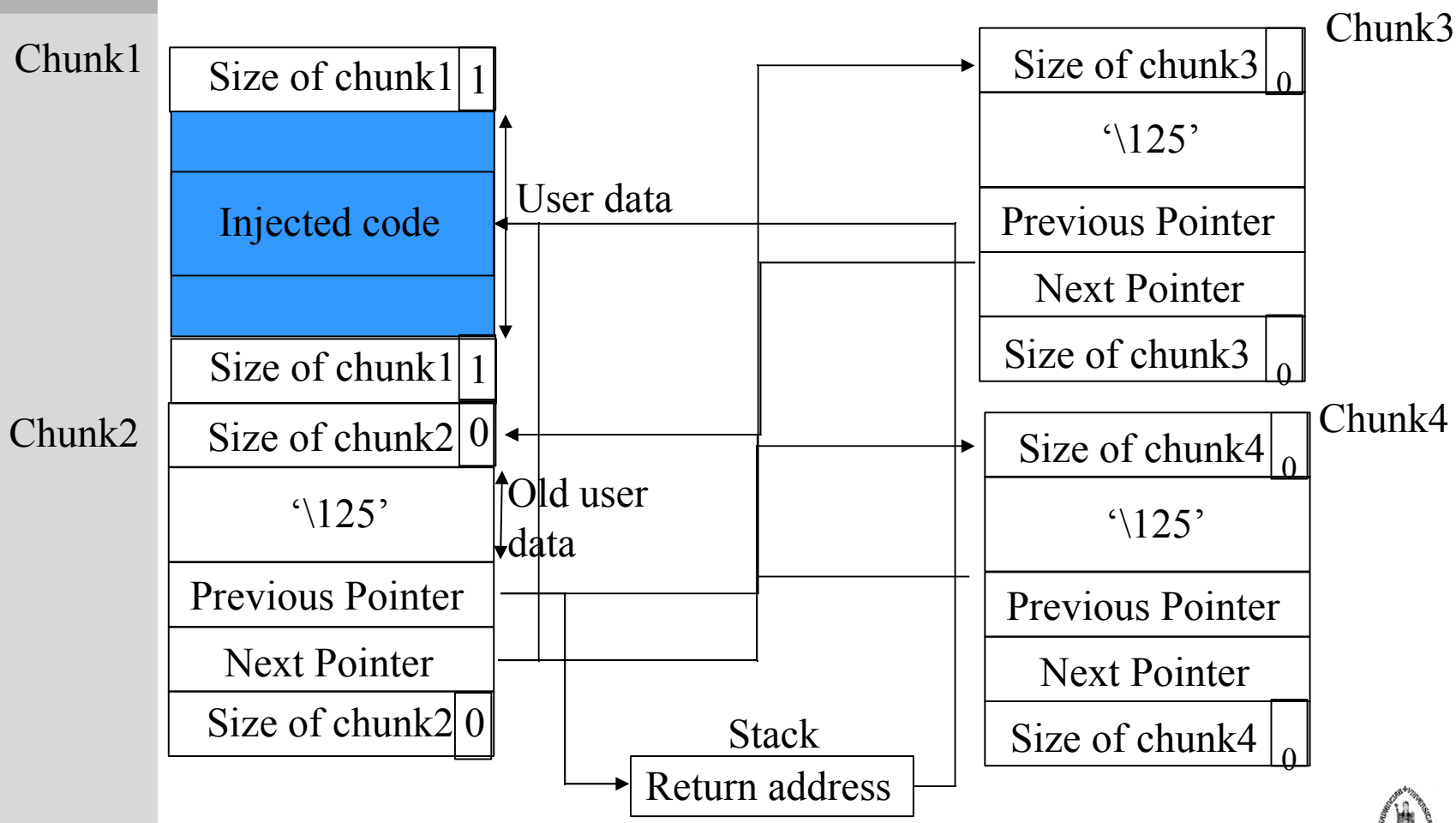
Overview

- Introduction
- Attacks
- Memory Allocators
  - Doug Lea's malloc (Linux)
  - CSRI/Quickfit
  - Phkmalloc (BSD)
  - Boehm's garbage collector
- A Safer Allocator
- Related Work
- Conclusion

CSRI/Quickfit



Heap Overflow (CSRI)



CSRI/Quickfit Conclusion

- Very similar to dlmalloc
- Vulnerable to heap overflow
- No double free possible: inuse bit checked
- No off by one/five possible: size at beginning compared with size at end

Overview

- Introduction
- Attacks
- Memory Allocators
  - Doug Lea's malloc (Linux)
  - CSRI/Quickfit
  - Phkmalloc (BSD)
  - Boehm's garbage collector
- A Safer Allocator
- Related Work
- Conclusion

Poul-Henning Kamp's malloc

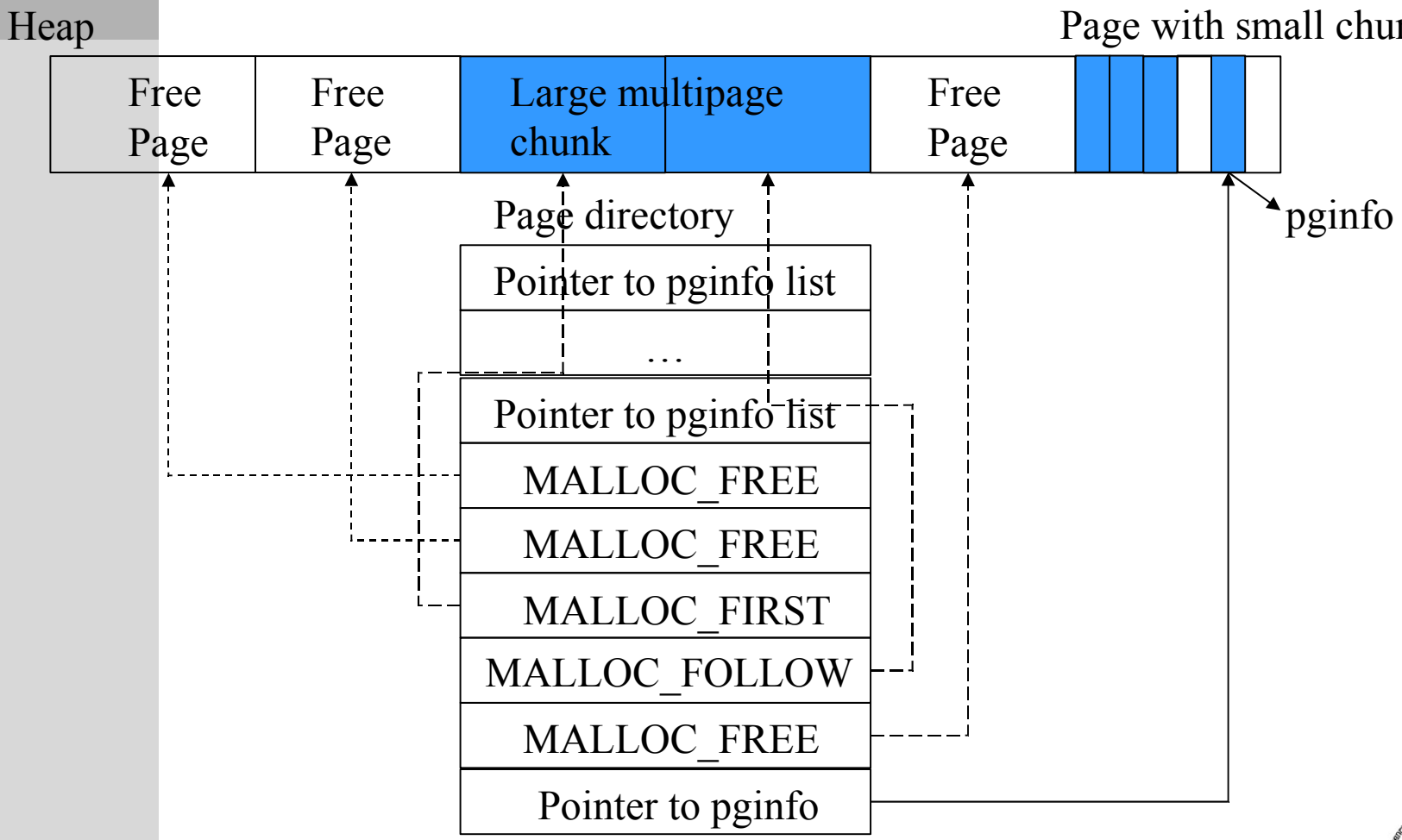
- Standard FreeBSD allocator (used in other BSDs as well).
- Takes advantage of the virtual memory system
- Tries to minimize accessed pages
- Makes sure that objects that are smaller or equal to a page do not span pages
- Two layers: page layer, chunk layer
- Attack on it first described by BBP

Phkmalloc: Page layer

- Manages allocation and freeing of pages
- Information about pages allocated to the heap are stored in a page directory
- First elements contain pointers to a linked list of page information for each possible size of chunks
- Other elements store information about particular pages

MALLOC_NOT_MINE	Not allocated by phk
MALLOC_FREE	Page is free
MALLOC_FIRST	Page is first in multipage object
MALLOC_FOLLOW	Following page (multipage object)

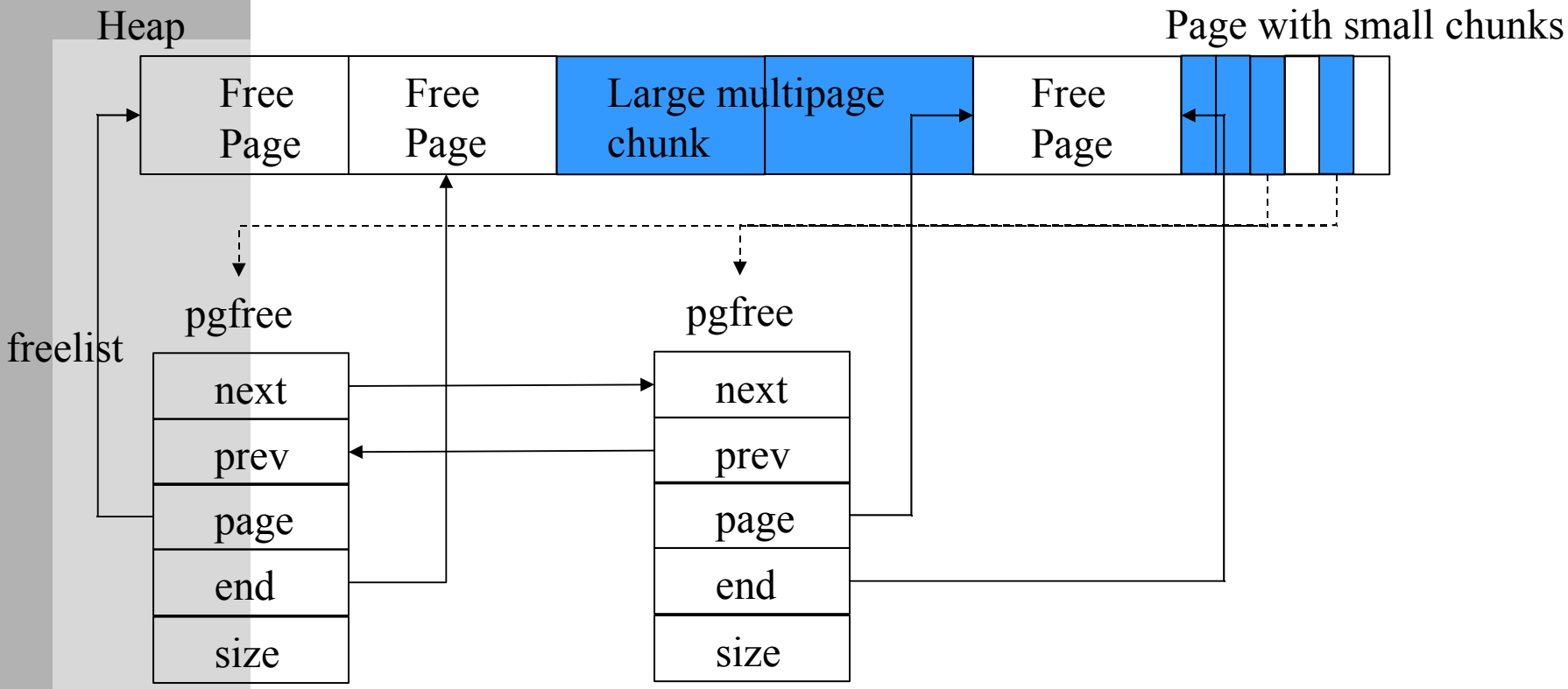
Phkmalloc: page directory



Phkmalloc: free pages

- Free pages stored in a sorted (address) doubly linked list
- Adjacent free pages are coalesced and only a pointer to the first page is kept
- Free page info (pgfree) is stored using malloc (allows swapping pages out)
- To increase performance a cache pointer is kept of a free pgfree

Phkmalloc: free pages



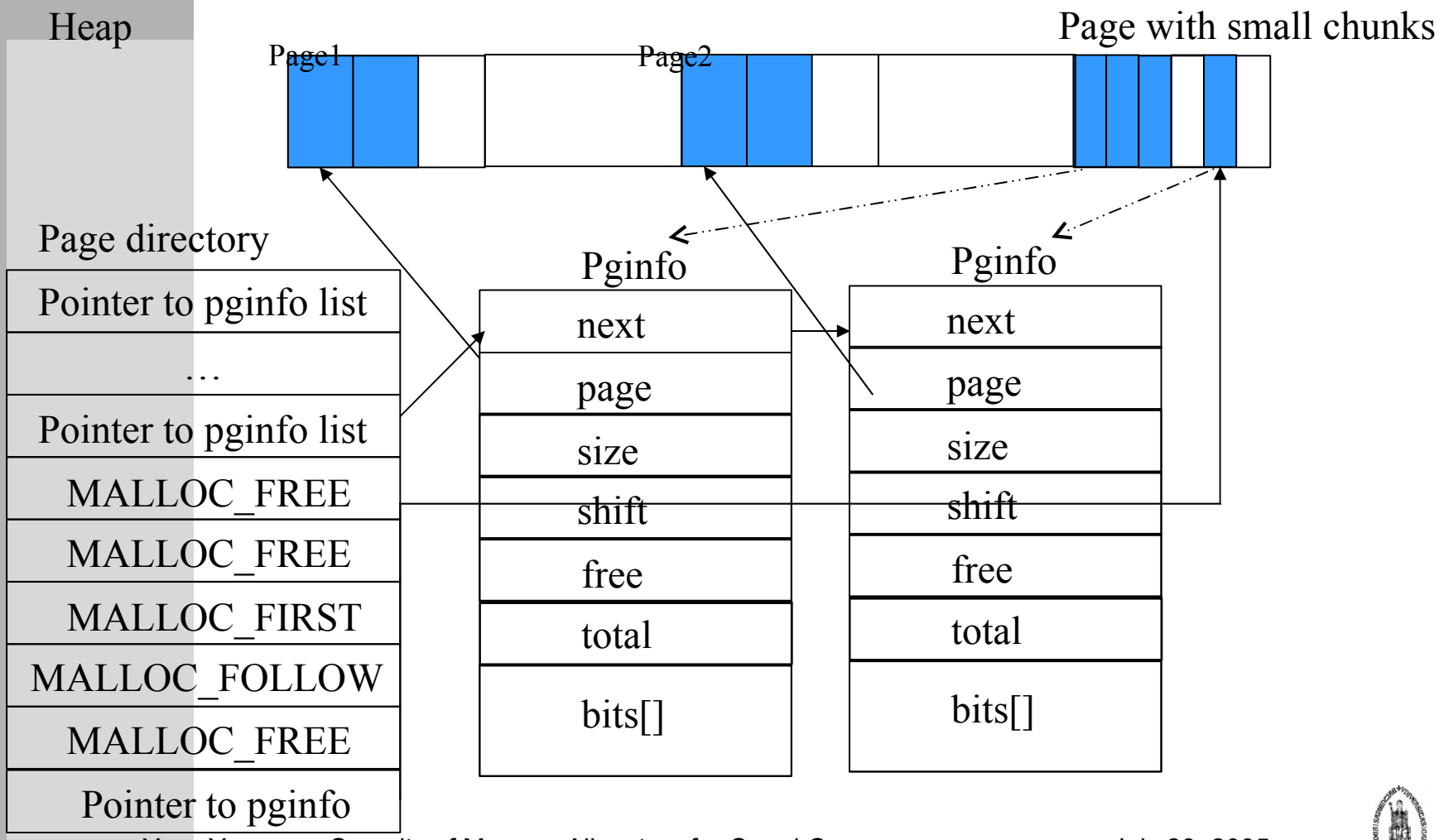
Phkmalloc: Chunk layer

- Three different types of chunks: small, medium large
- Large is larger than half a page
- Large object allocation: search for contiguous free pages, set MALLOC_FIRST and MALLOC_FOLLOW
- Small or medium chunks are rounded up to the next power of two
- Pages only contain chunks of the same size

Phkmalloc: pginfo

- Pginfo is used to describe a small or medium page
- Stored in the beginning of the page for small chunks
- Allocated using malloc for medium chunks
- The entry in the pagedir for small/medium pages points to pginfo

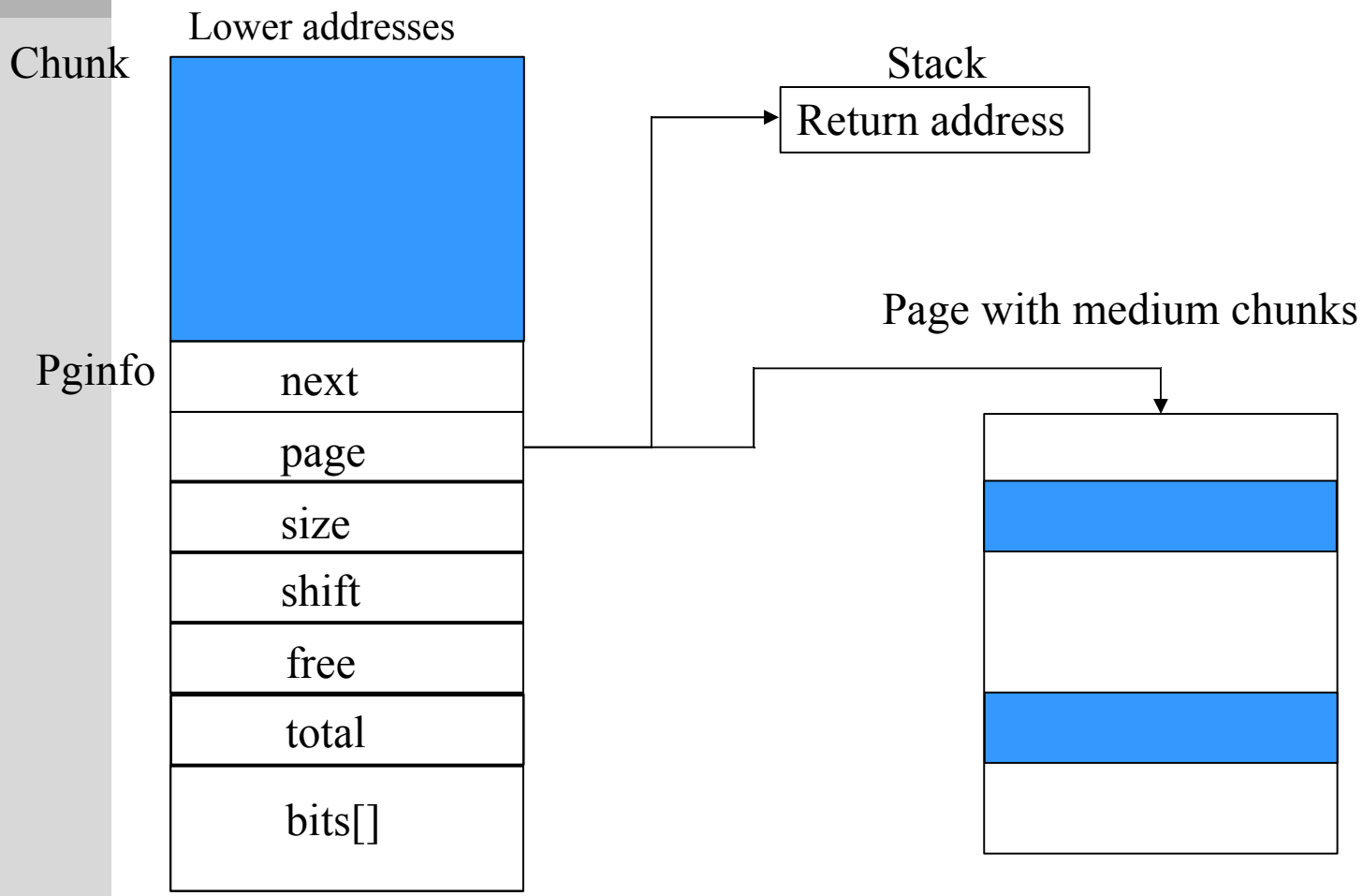
Phkmalloc: pginfo



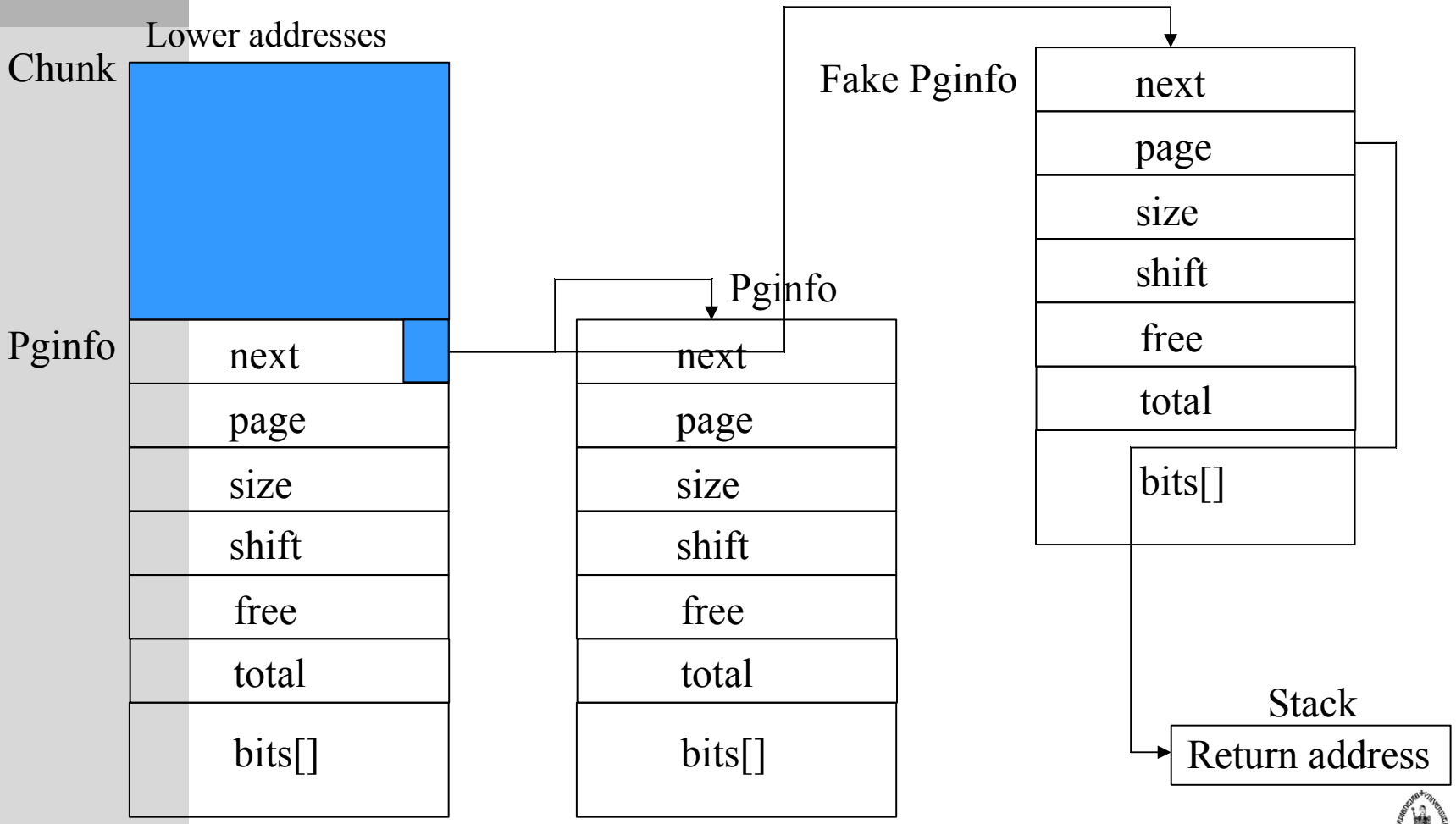
Heap overflow (phkmalloc)

- Pginfo (for medium chunks) and pgfree structs are stored together with normal chunks
- Can be overflowed
- We will demonstrate using pginfo
- Make page-field point to target memory
- Modify the bits[] array to make all chunks seem free
- When a chunk of that size is requested, the allocator will return the page-pointer as a

Heap overflow (phkmalloc)



Off by one (phkmalloc)



Phkmalloc conclusion

- Vulnerable to:
  - Heap overflow
  - Off by one
- Not vulnerable to double free: a check is done to see if the chunk is free or not

Overview

- Introduction
- Attacks
- Memory Allocators
  - Doug Lea's malloc (Linux)
  - CSRI/Quickfit
  - Phkmalloc (BSD)
  - Boehm's garbage collector
- A Safer Allocator
- Related Work
- Conclusion

Boehm's Garbage Collector

- Conservative garbage collector for C/C++
- Assumes values in memory are pointers if they look like pointers
- Automatically releases memory to the system when no longer needed
- No dangling pointer references are possible (unless programmer does explicit freeing)
- Uses mark and sweep

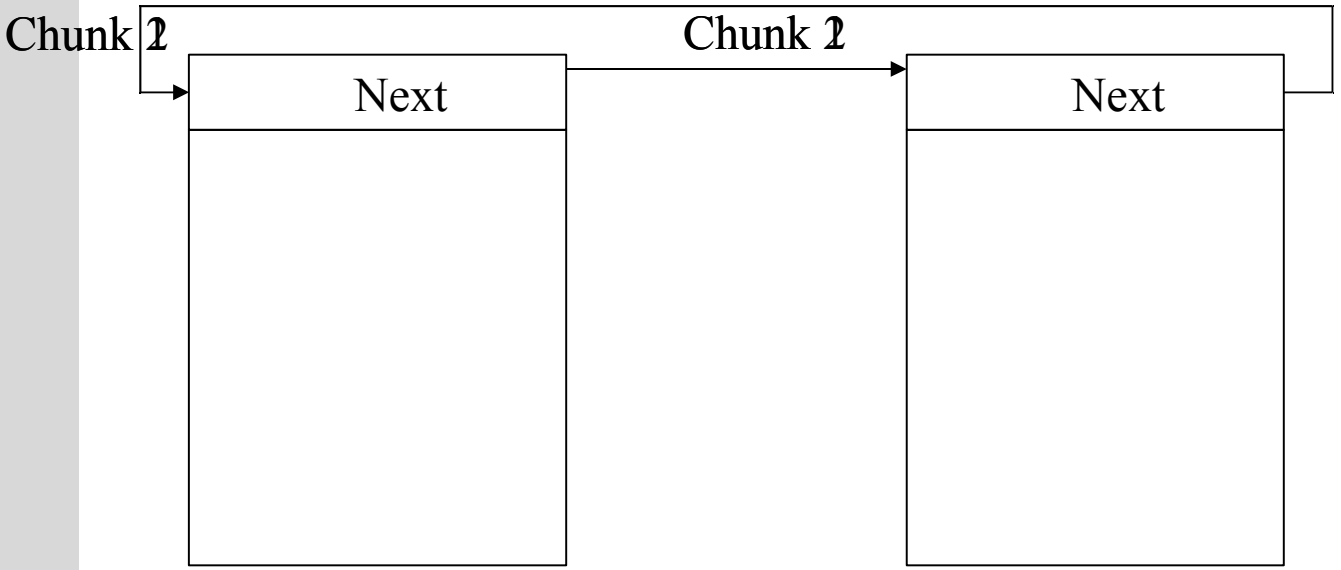
Boehm's Garbage Collector

- Makes a difference between small and large chunks
- Large chunks are larger than half of page size (IA32) and rounded up to page size
- Small chunks are allocated in pages and the page is divided in chunks of same size
- Allocated chunks only contain data
- Free chunks contain pointer to the next free chunk

Heap overflow (Boehm)

- If attackers can overflow a chunk, they can overwrite the next pointer
- An attacker can make the next pointer point to a target memory location
- Eventually the allocator will return the pointer's target as a valid chunk
- Usually an off by four attack

Double free (Boehm)



Boehm Conclusion

- Vulnerable to:
  - Heap overflow
  - Off by one-four
  - Double free: only if the programmer explicitly frees memory (usually not necessary)

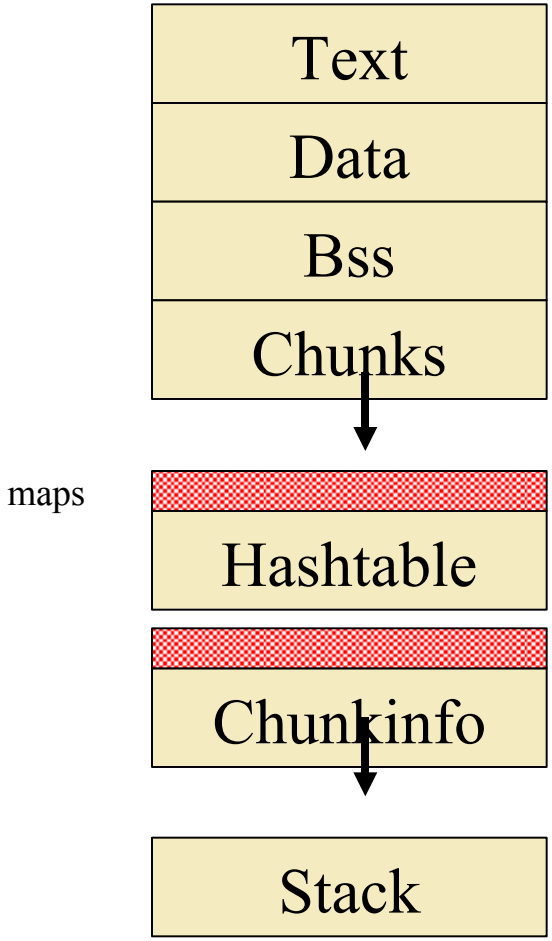
Overview

- Introduction
- Attacks
- Memory Allocators
- A Safer Allocator
- Related Work
- Conclusion

Design

- On most modern systems code and data are loaded into separate memory locations
- We apply the same to chunk information and chunks
- Chunk info is stored in separate contiguous memory
- This area is protected by guard pages
- A hashtable is used to associate chunks with chunkinfo
- The hashtable contains pointers to a linked list of chunk information accessed through the hashfunction

Modified memory layout

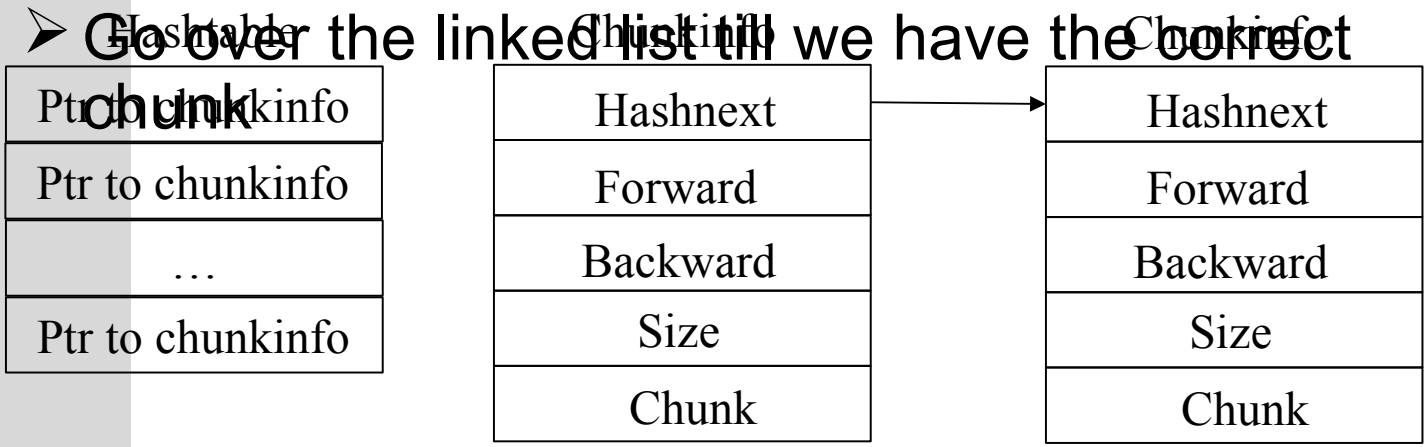


Dnmalloc: hashtable

- Hashtable is stored before the stack in a mmaped area big enough to hold it
- Each page is divided in 256 possible chunks (of 16 bytes, minimum chunk size)
- These chunks are separated into 32 groups of 8 chunks
- Each group has an entry in the hashtable, maximum 32 entries for every page
- One element of the group is accessed through a linked list of max. 8 elements

Dnmalloc: hashfunction

- To find a chunk's information from a chunk we do the following:
  - Substract the start of the heap from the chunk's address
  - Shift the result 7 bits to the right: gives us the entry in the hashtable
  - Go over the linked list till we have the correct chunk



Dnmalloc: Managing chunk information

- A fixed area is mapped for chunkinfos
- Free chunkinfos are stored in a linked list
- When a new chunkinfo is needed the first element in the free list is used
- If none are free a chunk is allocated from the map
- If the map is empty we map extra memory for it
- Chunk information is protected by

Dnmalloc performance overhead

Spec CPU2000 results for dlmalloc and dnmalloc
(13 runs on 8 identical pcs (P4 2.8ghz, 512mb) = 104 runs)

Program	Dlmalloc runtime	Dnmalloc runtime	Overhead percentage
gzip	253 +- 0	253.19 +- 0.01	0.08%
vpr	360.82 +- 0.15	361.25 +- 0.13	0.12%
gcc	153.94 +- 0.04	154.28 +- 0.04	0.22%
mcf	287.19 +- 0.07	290.16 +- 0.07	1.03%
crafty	253 +- 0	253.88 +- 0	0.27%
parser	346.94 +- 0.02	346.99 +- 0.05	0.01%
eon	771.09 +- 0.14	772.77 +- 0.11	0.22%
perlbmk	243.13 +- 0.03	252.11 +- 0.05	**3.69%**
gap	184.06 +- 0.02	184 +- 0	-0.03%
vortex	250 +- 0	248.96 +- 0.04	**-0.42%**
bzip2	361.62 +- 0.05	362.16 +- 0.07	0.15%
twolf	522.63 +- 0.41	527.5 +- 0.41	0.93%

Dnmalloc memory overhead

- Original dlmalloc overhead: +- 8 bytes per chunk
- Current overhead per program:
  - 4096 bytes for guard page for hashtable
- Overhead per 16384 chunks:
  - 4096 bytes for guard page for chunk info region
  - 4096 bytes for information page for chunk info region
- Overhead per chunk:
  - 20 bytes chunk information
  - 4 bytes hashtable entry

Dnmalloc memory overhead

Spec CPU2000 maximum memory usage for dlmalloc and dnmalloc

Program	Dlmalloc (MB)	Dnmalloc (MB)	Overhead percentage
gzip	180.37	180.38	**0.01%**
vpr	20.07	20.69	3.06%
gcc	81.02	81.09	0.09%
mcf	94.92	94.93	**0.01%**
crafty	0.84	0.86	1.49%
parser	30.08	30.09	0.04%
eon	0.33	0.35	6.84%
perlbmk	53.8	60.47	12.41%
gap	192.07	192.08	**0.01%**
vortex	60.17	61.32	1.91%
bzip2	184.92	184.93	**0.01%**
twolf	3.22	5.16	**60.08%**

Todo for dnmalloc

- When the free chunk information list is very large, it should release some of it (should reduce the memory overhead)
- Improve prev_chunkinfo lookup function (should improve performance)
- Should be done in the next few days (early January)

Overview

- Introduction
- Attacks
- Memory Allocators
- A Safer Allocator
- Related Work
  - Robertson et al. heap protector
  - Contrapolice
  - Glibc 2.3.5 integrity checks
- Conclusion

Robertson's heap protector

- Checksum stored in every chunk's header
- Checksum encrypted with a global read-only random value
- Checksum added when allocated, checked when freed
- Could be bypassed if information leaks exist
- Dlmalloc 2.8.x implements a slightly modified version of this

Overview

- Introduction
- Attacks
- Memory Allocators
- A Safer Allocator
- Related Work
  - Robertson et al. heap protector
  - Contrapolice
  - Glibc 2.3.5 integrity checks
- Conclusion

Contrapolice

- Protects chunks by placing canaries (random) before and after the chunk
- Before exiting from a copy function, it checks if the canary before matches the canary after
- Does not protect against non-copy overflows
- Could be bypassed if the canary value is leaked

Overview

- Introduction
- Attacks
- Memory Allocators
- A Safer Allocator
- Related Work
  - Robertson et al. heap protector
  - Contrapolice
  - Glibc 2.3.5 integrity checks
- Conclusion

Glibc 2.3.5 integrity checks

- Adds several integrity checks (unlink, free, realloc and other places):
  - Before unlink, check: p->fd->bk == p->bk->fd == p
  - Before free: pointer must be smaller than its negative size (and correctly aligned)
  - Various other checks for chunktypes that are handled differently
- Can be bypassed in some cases
- Attack discovered by Phantasmal Phantasmagoria
  - Describes 5 techniques for bypassing glibc integrity checks

Overview

- Introduction
- Attacks
- Memory Allocators
- A Safer Allocator
- Related Work
- Conclusion

Conclusion

- Many allocators ignore security issues
- Safer allocators are not necessarily much slower
- Our approach still has an important limitation: only chunk information is protected, not what is in the chunk
- This work is part of larger research where other important areas in memory are also separated from normal data (currently we're finishing up on a stack-based countermeasure)
- Which is part of my real research: a more methodical approach to designing countermeasures

Conclusion

- 3 papers associated with this talk:
  - 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
  - Yves Younan, Wouter Joosen and Frank Piessens. Applying machinemodel-aided countermeasure design to improve memory allocator security. 22nd Chaos Communication Congress, Berlin, Germany, December 2005, Chaos Computer Club.
  - Yves Younan, Wouter Joosen and Frank Piessens. A Methodology for Designing Countermeasures against Current and Future Code Injection Attacks, Proceedings of the Third IEEE International Information Assurance Workshop 2005 (IWIA2005), College Park, Maryland, U.S.A., March 2005, IEEE, IEEE Press.
- Dnmalloc implementation (tested on IA32 running 2.4 and 2.6) will be available tonight on http://www.fort-knox.org (also has other papers)

Thank you

- Questions?