Lec 17: Heap Hardening

IS561: Binary Code Analysis and Secure Software Systems

Sang Kil Cha

Heap Hardening

Recall: Heap Safety

A ***heap manager*** (a.k.a. heap allocator) helps organize memory objects, but memory corruption causes many troubles.

- Heap metadata corruption.
- Use-after-free vulnerabilities.

Question

How about designing a ***safe*** heap manager?

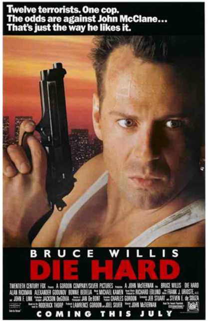
An Ideal World with Infinite Memory

- Every memory allocation returns a fresh new object.
- Every memory object is infinitely large, and objects do not overlap.
- No need to free objects.

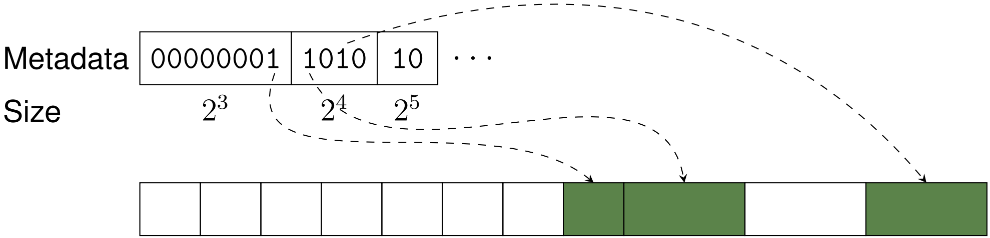
No heap metadata corruption, no UAF, no dangling pointers.

Secure Heap Allocators in Real World?

- DieHard: Probabilistic Memory Safety for Unsafe Programming Languages, ***PLDI 2006***
- DieHarder: Securing the Heap, ***CCS 2010***

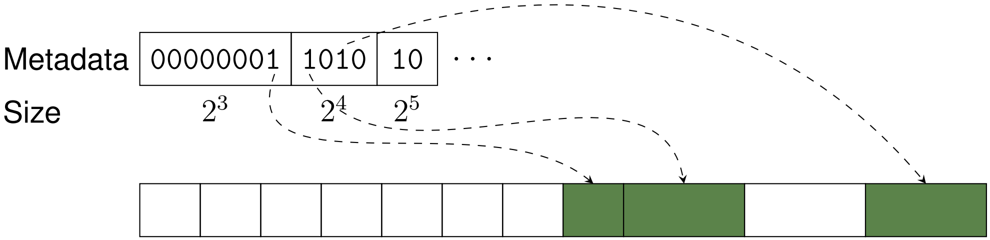


DieHard Design



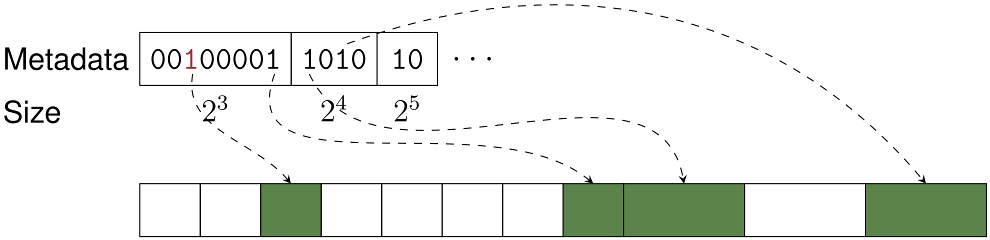
Heap metadata is separated from data. A bit in a bitmap represents one object: 0 means a freed slot, and 1 means an allocated object.

Randomized Allocation: `malloc(sz)`



1. Compute size class: `ceil(log sz) - 3`.
2. Randomly select a zero bit (which means a freed slot).

Randomized Allocation: `malloc(sz)`



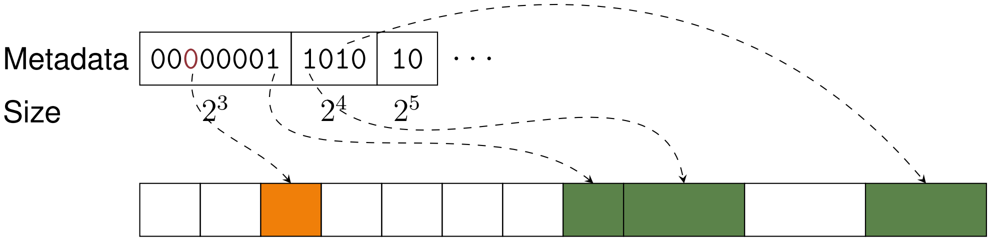
1. Compute size class: `ceil(log sz) - 3`.
2. Randomly select a zero bit (which means a freed slot).

DieHard Memory Allocation

- Allocation is fast: $O(1)$.
- Heap overflow will not overwrite the metadata.
- Heap overflow is non-deterministic: every overflow attempt will overwrite different memory objects[1].

[1] This is good and bad. Why?

Deallocation: `free(ptr)`



1. Check the bitmap to detect a double-free.
2. Modify the corresponding bit in the bitmap to zero.

Reflection on the Design of DieHard

- Security vs. performance trade-off.

Reflection on the Design of DieHard

- Security vs. performance trade-off.
  - Cache misses!

Reflection on the Design of DieHard

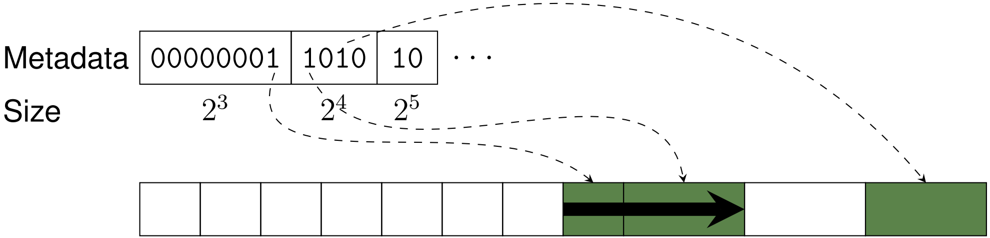
- Security vs. performance trade-off.
  - Cache misses!
- Still have a problem with ***uninitialized reads***.
  - Allocate a new object without initializing it.
  - Try to read previously written data from the object.

DieHarder Design

More ***secure*** than DieHard.

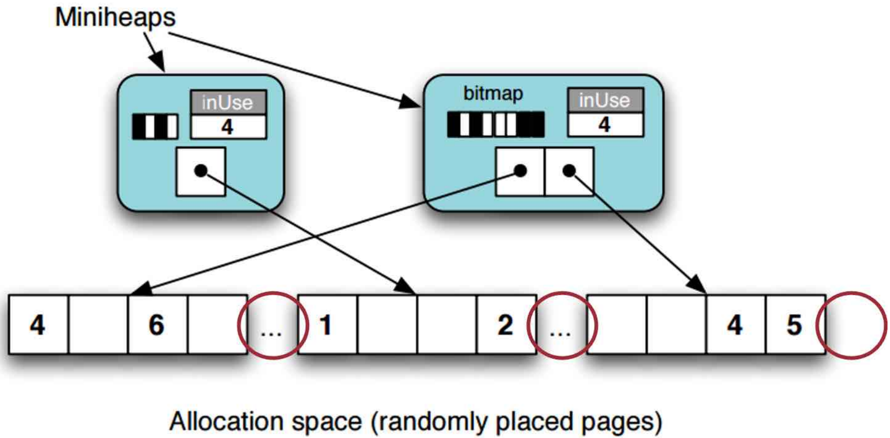
- Heap overflows can still corrupt memory objects. Can we make memory corruption ***less likely***?
- Uninitialized reads are problematic, can we prevent those attempts?

Problem #1: Memory Corruption



Corrupting adjacent objects.

Sparse Page Mapping



[2]

[2] Image from DieHarder: Securing the Heap, ***CCS 2010***.

Trade-Off: Security vs. Performance

Sparse page mapping increases the size of the page table.

Problem #2: Uninitialized Reads

- Freed objects keep original values.
- Old values can spray over the entire memory space.

Problem #2: Uninitialized Reads

- Freed objects keep original values.
- Old values can spray over the entire memory space.

Solution: destroy on free (= fill with random values)

Performance Overhead of DieHarder

- $0\times \sim 2\times$ overhead on SPEC CPU benchmark.
- Near zero performance overhead on Firefox
  - A sweet-spot of the security-performance trade-off.

Problem solved?

False Sharing Problem

Suppose o_1 and o_2 are used by two different threads T_1 and T_2, respectively. If o_1 and o_2 share the same cache line, writing to one object from a thread can cause cache misses in the other thread.

Most secure heap allocators do not consider this problem – every thread shares the same heap.

[3] FreeGuard: A Faster Secure Heap Allocator, ***CCS 2017***

False Sharing Problem

Suppose o_1 and o_2 are used by two different threads T_1 and T_2, respectively. If o_1 and o_2 share the same cache line, writing to one object from a thread can cause cache misses in the other thread.

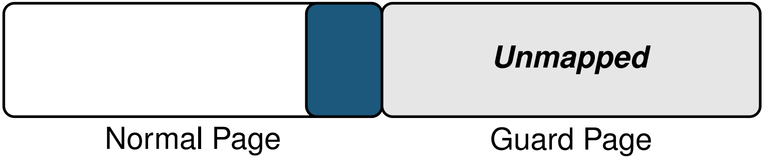
Most secure heap allocators do not consider this problem – every thread shares the same heap.

FreeGuard[3] addresses this problem by having a per-thread subheap design.

[3] FreeGuard: A Faster Secure Heap Allocator, ***CCS 2017***

An Extreme Case of Sparse Page Mapping

- Windows: PageHeap
- Linux: Electric Fence



Implication of PageHeap

Suppose we do ***not*** (or at least rarely) reuse memory while using PageHeap. This is also known as OTA (One Time Allocation) scheme.

Implication of PageHeap

Suppose we do ***not*** (or at least rarely) reuse memory while using PageHeap. This is also known as OTA (One Time Allocation) scheme.

> We can detect UAF bugs as well as heap memory corruption.

PageHeap Revisited

Prober: Practically Defending Overflows with Page Protection, ***ASE 2020***

- Can we apply the idea of PageHeap on a reduced scope?
- Key intuition: overflowing objects are typically related to arrays.
- Put array-related objects to a separate space with the PageHeap protection!

PageHeap Revisited (Again)

Preventing Use-After-Free Attacks with Fast Forward Allocation, ***USENIX Security 2021***.

- Discuss several practical issues, such as VMA exhaustion.
- But still inefficient for many real-world applications especially with many short-lived objects (frequent malloc/free calls). ***Fragmentation*** is a big issue.
- More recent advances with kernel support[4]
- Can only handle UAF bugs.

[4]BUDAlloc: Defeating Use-After-Free Bugs by Decoupling Virtual Address Management from Kernel, ***USENIX Security 2024***

Key Takeaway

Performance vs. Security.

Question?

Exercise: Try DieHard

Download DieHard from `https://github.com/emeryberger/DieHard`, and use it. Create a toy program that calls `malloc`s and `free`s, and attach GDB to its process to see how the allocator behaves.